Link : https://hackmd.io/72mN4xn3RU6E0rmdPABgzQ?view

(The hackmd is provided above since I use screenshot to keep the layout, and some links can't be accessed)

hw5

Gaussian Process

Code

- **Task1**
  - Rational quadratic kernal
    - $k_{RQ}(x, x')$ = $(1 + \frac{\|x-x'\|^2}{2\alpha l^2})^{-\alpha}$
    - ```
      def k_RQ(xi, xj, l, a):
          # l : length scale with l > 0
          # a(ahpla) : scale mixture param with α > 0
          # (1 + (xi - xj)^2) / (2αl^2)^(-α)
          return (1 + cdist(xi, xj, 'sqeuclidean')/(2 * a * (l ** 2))) ** (-a)
      ```
  - Marginal likelihood
    - $p(y) = \int p(y|f)p(f)df = N(y|0, C)$
      - Yet, there are noise $\epsilon \sim N(\cdot|0, \beta^{-1})$
      - Thus, $C(x_n, x_m) = k(x_n, x_m) + \beta^{-1}\delta_{nm}$
      - δ = 1 if n==m, otherwise 0, so we rewrite $\beta^{-1}\delta_{nm}$ to $\beta^{-1}I$
    - ```
      def cov(l, alpha):
          cov_matix = k_RQ(X, X, l, alpha)
          noise = (1 / beta) * np.identity(X.shape[0])
          return (cov_matix + noise)
      ```
  - Goal : Conditional Distribution $p(y^*|y) \sim N(\mu(x^*), \sigma(x^*)^2)$
    - $\mu(x^*) = k(x, x^*)^T C^{-1} y$
    - $\sigma^2(x^2) = k^* - k(x, x^*)^T C^{-1} k(x, x^*)$
    - $k^* = k(x^*, x^*) + \beta^{-1}$
    - ```
      def Gaussian_Process(l=1.0, alpha=1.0):
          # compute C, k(x, x*), and k*
          C = cov(l, alpha)
          k_x_xstar = k_RQ(X, Xstar, l, alpha)
          kstar = k_RQ(Xstar, Xstar, l, alpha) + 1 / beta

          # p(y*|y) ~ N(μ, σ^2)
          mean = k_x_xstar.T @ (np.linalg.inv(C)) @ Y
          var = kstar - (k_x_xstar.T @ (np.linalg.inv(C) @ k_x_xstar))

          # plot the figure
          visualization(mean, var)
          return
      ```
      ```

- Visualization
  - Use z-score to mark 95% confidence interval where z = $\frac{x-\mu}{\sigma}$ = $\pm$ 1.96
  -
```
def visualization(mean, var):
    # z = ±1.96 = (x-m)/var => x = m ±1.96 * var
    upper = mean + 1.96 * var.diagonal().reshape(-1, 1)
    lower = mean - 1.96 * var.diagonal().reshape(-1, 1)

    plt.xlim(-60, 60)
    plt.ylim(max(upper) + 2, min(lower) - 2)
    # Mark 95% interval
    plt.fill_between(Xtest, upper.ravel(), lower.ravel(), color='thistle')
    # Draw a line to represent mean of f
    plt.plot(Xtest, mean, 'royalblue')
    #Show all training data points
    plt.scatter(X, Y, s=10, c='r')
    plt.show()
```

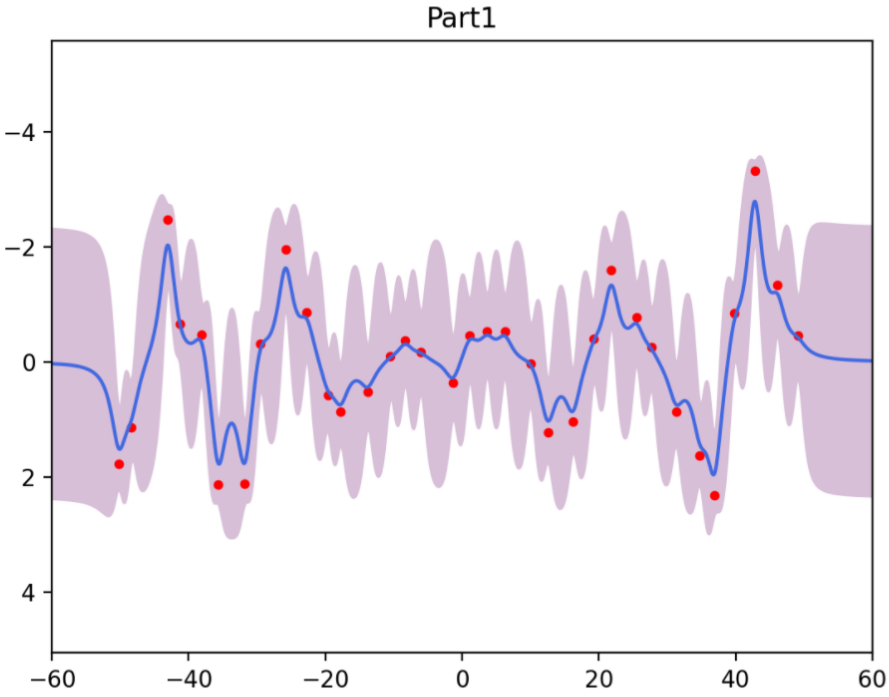
- **Task2 : Optimize the kernal**
  - Marginal likelihood $p(y|\theta) \sim N(y|0, C_\theta)$
  - Negative marginal log likelihood J = $\ln p(y|\theta) = \frac{1}{2}[N\ln(2\pi) + y^T C_\theta^{-1} y + \ln|C_\theta|]$
    - Use J to transfer a maximization problem into minimization problem.
    - Call scipy.optimize.minimize to find optimal θ
      - θ_{new} = `minimize(objective_function, theta)`
      -
```
def objective_function(theta):
    # Objective function J = -ln(p(y|θ) = 0.5[Nln(2π)+ yT(C-^1)y + ln|Cθ|]
    # θ = (l, α)
    N = X.shape[0]
    YT = Y.ravel().T
    C = cov(theta[0], theta[1])

    obj_f = 0.5 * (N * np.log(2 * np.pi) +
                   (YT @ np.linalg.inv(C) @ Y) +
                   np.log(np.linalg.det(C)))
    return obj_f[0]
```

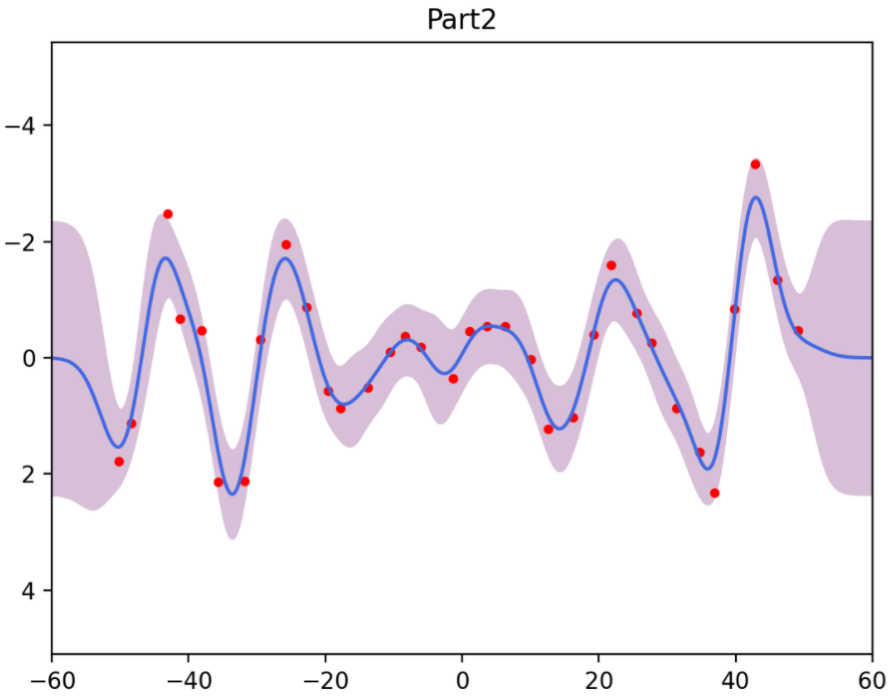
    - Now, we can try gaussian process again using this updated hyperparameter θ_{new}

Experiments

- **Part1**
  - l = 1.0, α = 1.0



- **Part2**
  - l = 2.9705239253258946, α = 1026.4899617040603



Observations and Discussion

- Comparison between part1 & 2
  - Optimization makes the 95% interval smaller, so the prediction is more precise.
  - Optimization makes the curve smoother, rather than overfitting.
- Observation of confidence interval
  - Confidence interval(variance) is quite large at which without trainging data.
    - This is because the prediction is based on training data
    - In contrast, small confidence interval is usaually due to the presence of training data.

SVM on MNIST

Code

- **Task1**
  - Wrap libsvm functions in svm()

```
if __name__ == '__main__':
    # Task 1
    svm('linear')
    svm('polynomial')
    svm('RBF')
```

  - Refer to document to see the meanings of parameters

```
-t kernel_type : set type of kernel function (default 2)
    0 -- linear: u'*v
    1 -- polynomial: (gamma*u'*v + coef0)^degree
    2 -- radial basis function: exp(-gamma*|u-v|^2)
```

  - svm() : self-defined wrapper function
    -

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

def svm(k):
    print(f'kernel_type : {k}')

    start = time.time()

    param = svm_parameter(f'-t {kernel[k]}')
    prob  = svm_problem(Y_train, X_train)
    model = svm_train(prob, param)
    _, p_acc, _ = svm_predict(Y_test, X_test, model)

    end = time.time()
    print("Time: %0.2f seconds." % (end - start))
    print()
```

- **Task2**
  - In soft-margin SVC, penalty C can be tuned.

```
def grid_search_on_c(arg, max_acc):
    best_c = 1e-1
    for c in [1e-2, 1e-1, 1e0, 1e1, 1e2]:
        param = svm_parameter(arg.format(c))
        prob  = svm_problem(Y_train, X_train)
        p_acc = svm_train(prob, param)
        if p_acc > max_acc:
            max_acc = p_acc
            best_c = c
    return max_acc, best_c
```

  - grid_search()
    - Idea
      - In the guide, it says that since doing a complete grid-search may still be time-consuming, it's recommended to use a coarse grid first.
      - After identifying a "better" region on the grid, a finer grid search on that region can be conducted.
    - Implement
      - Loops are used to try on various pairs of values to find.
      - The params for different types of kernal are as follows :

```
-c cost : set the parameter C of C-SVC, epsilon-SVR, and nu-SVR (defa
-d degree : set degree in kernel function (default 3)
-g gamma : set gamma in kernel function (default 1/k)
-r coef0 : set coef0 in kernel function (default 0)
-v : calling svm_train with -v n will return the best n-fold cross-va
```

      - Linear kernal : -c C
      - polynomial kernal $(\gamma x_i^T x_j + r)^d, \gamma > 0$: -c C, -g γ, -r d
      - RBF kernal : $exp(-\gamma \|x_i - x_j'\|^2)$: -c C, -g γ
    - Code
      -

```
def grid_search(k):

    fold = 5
    print(f'kernel_type : {k}')

    time_start = time.time()
    max_acc = 0.0

    if k == 'linear':
        arg = f'-t {kernel[k]} -c ' + '{} ' + f'-v {fold} -q'
        max_acc, best_c = grid_search_on_c(arg, max_acc)
        best_params = {'C': best_c}

    elif k == 'polynomial':
        for degree in range(1, 3):
            for gamma in [1e0, 1e1]:
                for coef0 in [1e0, 1e1]:
                    arg = f'-t {kernel[k]} -c ' + '{} ' + f'-g {gamma} -d {degree} -r {coef0} -v {fold} -q'
                    local_max_acc, best_c = grid_search_on_c(arg, max_acc)
                    if local_max_acc > max_acc:
                        max_acc = local_max_acc
                        best_params = {'degree':degree,'gamma':gamma,'coef0':coef0,'C':best_c}

    elif k == 'RBF':
        for gamma in [1e-3, 1e-2, 1e-1]:
            arg = f'-t {kernel[k]} -c ' + '{} ' + f'-g {gamma} -v {fold} -q'
            local_max_acc, best_c = grid_search_on_c(arg, max_acc)
            if local_max_acc > max_acc:
                max_acc = local_max_acc
                best_params = {'gamma':gamma,'C':best_c}

    time_end = time.time()

    print(f'Best acc : {max_acc}')
    print(f'Best Params : {best_params}')
    print("Time: %0.2f seconds." % (time_end-time_start))
    print()
```

        - Note
            - When using polynomial method, I tried γ on [1e-2, 1e-1, 1e0, 1e1, 1e2]at first, but the accuracy is low at the beginning(about 20%), and the process is extremely time-consuming.
            - Then I turned to another range [1e-1, 1e0, 1e1], and about 98% accuracy was achieved. The same work were done on degree, coef0 at the same time.

- **Task3**
    - param
        - svm_param
            - `-t 4 -- precomputed kernel (kernel values in training_set_file)`
        - svm_problem
            - `isKernel=True for precomputed kernel`
    - Code
        - kernal functions (fomula):
            - linear kernal : $x_i x_j^T$
            - RBF kernal : $exp(-\gamma \|x_i - x_j'\|^2)$
            -
```
def linear_kernel(xi, xj):
    return xi @ xj.T

def RBF_kernel(u, v, gamma):
    return np.exp(-gamma * cdist(u, v, 'sqeuclidean'))
```
        - Do grid search on the new model as task2
            -
```
def linear_kernel(xi, xj):
    return xi @ xj.T

def RBF_kernel(u, v, gamma):
    return np.exp(-gamma * cdist(u, v, 'sqeuclidean'))

def svm_combined_kernel():
    fold = 5
    max_acc = 0.0
    time_start = time.time()

    for gamma in [1e-3, 1e-2, 1e-1, 1e0, 1e1]:
        # Build a new kernal by combining linear and rbf kernal
        X_train_new = linear_kernel(X_train, X_train) + RBF_kernel(X_train, X_train, gamma)
        X_train_new = np.hstack((np.arange(1, len(X_train)+1).reshape(-1, 1), X_train_new))
        for c in [1e-2, 1e-1, 1e0, 1e1, 1e2]:
          # train the svm
            arg = f'-t 4 -c {c} -g {gamma} -v {fold} -q'
            param = svm_parameter(arg)
            prob  = svm_problem(Y_train, X_train_new, isKernel=True)
            p_acc = svm_train(prob, param)
            if p_acc > max_acc:
                max_acc = p_acc
                best_params = {'gamma':gamma,'C':c}

    time_end = time.time()
    print(f'Best acc : {max_acc}')
    print(f'Best Params : {best_params}')
    print("Time: %0.2f seconds." % (time_end-time_start))

if __name__ == '__main__':
    svm_combined_kernel()
```

Experiments

- **Task1**
  -
```
kernel_type : linear
Accuracy = 100% (5000/5000) (classification)
Time: 6.60 seconds.

kernel_type : polynomial
Accuracy = 34.34% (1717/5000) (classification)
Time: 55.99 seconds.

kernel_type : RBF
Accuracy = 96.88% (4844/5000) (classification)
Time: 14.18 seconds.
```
  -
- **Task2**
  -
```
----------------------------------------------

kernel_type : linear
Cross Validation Accuracy = 97.04%
Cross Validation Accuracy = 97.12%
Cross Validation Accuracy = 96.3%
Cross Validation Accuracy = 96.3%
Cross Validation Accuracy = 95.88%
Best acc : 97.11999999999999
Best Params : {'C': 0.1}
Time: 57.34 seconds.
```
  -
```
--------------------------------------------

kernel_type : polynomial
Cross Validation Accuracy = 96.98%
Cross Validation Accuracy = 96.98%
Cross Validation Accuracy = 96.52%
Cross Validation Accuracy = 96.18%
Cross Validation Accuracy = 96.38%
Cross Validation Accuracy = 96.94%
Cross Validation Accuracy = 96.48%
Cross Validation Accuracy = 96.24%
Cross Validation Accuracy = 96.66%
Cross Validation Accuracy = 96.32%
Cross Validation Accuracy = 96.8%
Cross Validation Accuracy = 96.2%
Cross Validation Accuracy = 95.94%
Cross Validation Accuracy = 96.22%
Cross Validation Accuracy = 96.1%
Cross Validation Accuracy = 96.96%
Cross Validation Accuracy = 96.04%
Cross Validation Accuracy = 96.2%
Cross Validation Accuracy = 96.16%
Cross Validation Accuracy = 96.28%
Cross Validation Accuracy = 98.14%
Cross Validation Accuracy = 98.14%
Cross Validation Accuracy = 97.96%
Cross Validation Accuracy = 98%
Cross Validation Accuracy = 98.24%
Cross Validation Accuracy = 98.02%
Cross Validation Accuracy = 98.12%
Cross Validation Accuracy = 97.94%
Cross Validation Accuracy = 98%
Cross Validation Accuracy = 97.98%
Cross Validation Accuracy = 98.04%
Cross Validation Accuracy = 98.04%
Cross Validation Accuracy = 98.12%
Cross Validation Accuracy = 98.08%
Cross Validation Accuracy = 97.98%
Cross Validation Accuracy = 98.04%
Cross Validation Accuracy = 98.1%
Cross Validation Accuracy = 97.96%
Cross Validation Accuracy = 98.16%
Cross Validation Accuracy = 98.26%
Best acc : 98.26
Best Params : {'degree': 2, 'gamma': 10.0, 'coef0': 10.0, 'C': 100.0}
Time: 475.81 seconds.
```
  -
```
----------------------------------------------

kernel_type : RBF
Cross Validation Accuracy = 81.08%
Cross Validation Accuracy = 92.44%
Cross Validation Accuracy = 96.12%
Cross Validation Accuracy = 97.18%
Cross Validation Accuracy = 97.28%
Cross Validation Accuracy = 92.34%
Cross Validation Accuracy = 96.46%
Cross Validation Accuracy = 97.7%
Cross Validation Accuracy = 98.22%
Cross Validation Accuracy = 98.44%
Cross Validation Accuracy = 48.82%
Cross Validation Accuracy = 53.22%
Cross Validation Accuracy = 91.98%
Cross Validation Accuracy = 92.38%
Cross Validation Accuracy = 92.46%
Best acc : 98.44000000000001
Best Params : {'gamma': 0.01, 'C': 100.0}
Time: 1042.24 seconds.
```

- **Task3**
  - ```
    Cross Validation Accuracy = 96.9%
    Cross Validation Accuracy = 96.9%
    Cross Validation Accuracy = 96.06%
    Cross Validation Accuracy = 96.44%
    Cross Validation Accuracy = 96.32%
    Cross Validation Accuracy = 96.98%
    Cross Validation Accuracy = 97.14%
    Cross Validation Accuracy = 96.32%
    Cross Validation Accuracy = 96.16%
    Cross Validation Accuracy = 96.2%
    Cross Validation Accuracy = 97.04%
    Cross Validation Accuracy = 97%
    Cross Validation Accuracy = 96.54%
    Cross Validation Accuracy = 96.36%
    Cross Validation Accuracy = 96.68%
    Cross Validation Accuracy = 96.88%
    Cross Validation Accuracy = 96.92%
    Cross Validation Accuracy = 96.34%
    Cross Validation Accuracy = 96.3%
    Cross Validation Accuracy = 96.48%
    Cross Validation Accuracy = 97.1%
    Cross Validation Accuracy = 97%
    Cross Validation Accuracy = 96.52%
    Cross Validation Accuracy = 96.5%
    Cross Validation Accuracy = 96.58%
    Best acc : 97.14
    Best Params : {'gamma': 0.01, 'C': 0.1}
    Time: 652.47 seconds.
    ```

Observations and Discussion

- **Task1**
  - Let's have a summary first.
	linear	polynomial	RBF
Test-Acc(%)	100	34.34	96.88
Time(s)	6.6	55.99	14.18
  - Comparison
    - Linear kernal has the best accuracy and best performance.
    - RBF kernal also has a nice accuracy and acceptable performance
    - Polynomial kernal has the worst accracy and performance.
  - Discussion
    - The data seems linearly seperable, so the linear kernal is good enough for classificaiton.
    - In the guide, it is said that "after searching the (C, γ) space, RBF will performs at least as good as linear".
    - However, before tunung the hyperparameter, RBF may perform worse than linear kernal.
    ```

- **Task2**
  - Let's have a summary first.
	linear	polynomial	RBF
CV-Acc(%)	97.12	98.26	98.44
Test-Acc(%)	99.2	100	100
params	-c 0.01	-c 100 -g -10 d 2 -r 10	-g 0.01 -c 100
Time(s)	57.34	475.81	1042.24
  - About the performance of polynomial and RBF kernal
    - Since polynomial and RBF kernal are too time-consuming, I have decreased the range of grid to [1e0, 1e1] and [1e-3, 1e-2, 1e-1] respectively after trying on a larger range(i.e., [1e-2, 1e-1, 1e0, 1e1,1e2]).
    - Polynomial and RBF kernal require more params, so it's reasonable that they are slow when the params are not picked properly.
  - About linear and RBF
    - After tuning, RBF performs better than either linear or polynomial kernal, but it's time-consuming.
    - Thus, if the data is linearly seperable, using the linear kernel is good enough and moreover, the model will run faster. Another reason is that if the number of features is large, mapping data to a high-dim space doesn't improve the performance much.
    - However, to persue the best accuracy, RBF is still a nice and more genral method.
- **Task3**
  - Let's have a summary first.
	linear	linear+RBF	RBF
CV-Acc(%)	97.12	97.14	98.44
Test-Acc(%)	99.2	99.26	100
params	-c 0.01	-c 0.1 -g 0.01	-g 0.01 -c 100
param range	range of c are same on the 3 model	g=[1e-3, 1e-2, 1e-1, 1e0, 1e1]	g=[1e-3, 1e-2, 1e-1]
Time(s)	57.34	652.47	1042.24
  - Obervation
    - The accuracy of this new combined model is between linear and RBF.
    - The performance of this new combined model is also between linear and RBF.
  - Discussion
    - The new model seems to be a compromise method, especially when one asks for performance but doesn't care too much about accuracy.
    - In this case, since the linear kernal is good enough, I think the other 2 models are still not necessary.