Gaussian Process

- **Code for task 1**

```
class GaussianProcess:
    def __init__(self, train_X, train_Y, beta=5):
        self.train_x = train_X
        self.train_y = train_Y
        self.num_of_test = 1000
        self.beta = beta
        self.alpha = 1
        self.l = 1
```

  - First, I construct the **GaussianProcess** class and implement some methods within the class.
  - Some *variables* in this class include:
    - *train_x*: the x-coordinates of training data points.
    - *train_y*: the y-coordinates of training data points.
    - *num_of_test*: the number of points to predict the function f within the range $x \in [-60, 60]$.
    - *beta*: the noise parameter.
    - *alpha*: the scale mixture parameter for the rational quadratic kernel.
    - *l*: the length scale parameter for the rational quadratic kernel.

```
def RQ_kernel(self, x1, x2):
    alpha, l = self.alpha, self.l
    dis = np.sum(x1**2, axis=1).reshape(-1, 1) + np.sum(x2**2, axis=1) - 2 * np.dot(x1, x2.T)
    return (1 + dis / (2 * alpha * l * l))**(-alpha)
```

  - **RQ_kernel**: the method of *implementing the Rational Quadratic Kernel* (Reference (https://scikit-learn.org/stable/modules/generated/sklearn.gaussian_process.kernels.RationalQuadratic.html)).
  - The kernel is given by: $k(x_i, x_j) = \left(1 + \frac{d(x_i, x_j)^2}{2\alpha l^2}\right)^{-\alpha}$.

```
def cal_cov(self, x):
    dim = len(x)
    return self.RQ_kernel(x, x) + np.eye(dim) * (1 / self.beta)
```

  - **cal_cov**: the method of *calculating the covariance*.
  - The covariance matrix is given by: $C(x_n, x_m) = k(x_n, x_m) + \beta^{-1}\delta_{nm}$.

```python
def predict(self):
    test_x = np.linspace(-60, 60, self.num_of_test).reshape(-1, 1)

    # calculate the covariance and kernel
    C = self.cal_cov(self.train_x)
    K = self.RQ_kernel(self.train_x, test_x)

    # calaulate mean and variance
    mu = mul(mul(K.T, inv(C)), self.train_y)
    k_star = self.RQ_kernel(test_x, test_x) + (1 / self.beta)
    var = k_star - mul(mul(K.T, inv(C)), K)
```

- **predict**: the method of *predicting and calculating the results*.
- With the covariance of the training data (prior), I can easily derive the posterior by calculating the kernel as follows: (x is training data and x^* is testing data)
  - $\mu(x^*) = k(x, x^*)^T C^{-1} y$
  - $k^* = k(x^*, x^*) + \beta^{-1}$
  - $\sigma^2(x^*) = k^* - k(x, x^*)^T C^{-1} k(x, x^*)$
- Note that the hyper-parameters in kernel function are (default) set to $\alpha = 1$ and $l = 1$.

```python
def plot(mu, var):
    plt.style.use('fast')
    fig = plt.figure(figsize=(6,4))
    axe = fig.add_subplot()
    axe.set_xlim([-60, 60])
    axe.set_ylim([-5, 5])
    axe.scatter(self.train_x, self.train_y, c='blue')

    m = mu.ravel()
    x = np.linspace(-60, 60, self.num_of_test)
    CI_half = 1.96 * np.sqrt(np.diag(var))

    axe.plot(x, m, 'red')
    axe.fill_between(x, m + CI_half, m - CI_half, facecolor='green', alpha=0.18)
    plt.show()
```

- **plot**: the method of *showing the prediction results* within the range $[-60, 60]$, and drawing the red line to denote the μ^* for any test point x^*.
- The 95% confidence interval bound is given by: $1.96 \times \text{standard deviation}$.
- Then, I use *fill_between* function to shade the area between the bounds.

```
if __name__=='__main__':
    X, Y = load_data("./data/input.data")

    GPR = GaussianProcess(X, Y)
    GPR.run()
```

```
def run(self):
    # predict with initial parameter setting (alpha = 1 and l = 1)
    self.predict()
```

- I construct the GaussianProcess in **main**, and run the **predict** method for Task 1.

- **Code for task 2**
  - Now, we have two hyper-parameters α and l, and I want to find the best parameters by minimizing the negative log likelihood (maximizing the likelihood). In practical, the parameters influence the covariance matrix.

```
def opt(self):
    def negative_log_likelihood_loss(params):
        self.alpha, self.l = params[0], params[1]
        C = self.cal_cov(self.train_x)
        neg_log_likelihood = 0.5 * np.linalg.slogdet(C)[1] \
                             + 0.5 * mul(mul(self.train_y.T, inv(C)), self.train_y) \
                             + 0.5 * len(self.train_x) * np.log(2 * np.pi)
        return neg_log_likelihood.ravel()

    res = minimize(negative_log_likelihood_loss,
                   [1, 1],
                   bounds=((1e-5, 1e5), (1e-5, 1e5)))

    self.alpha = res.x[0]
    self.l = res.x[1]
    print(self.alpha, self.l)
```

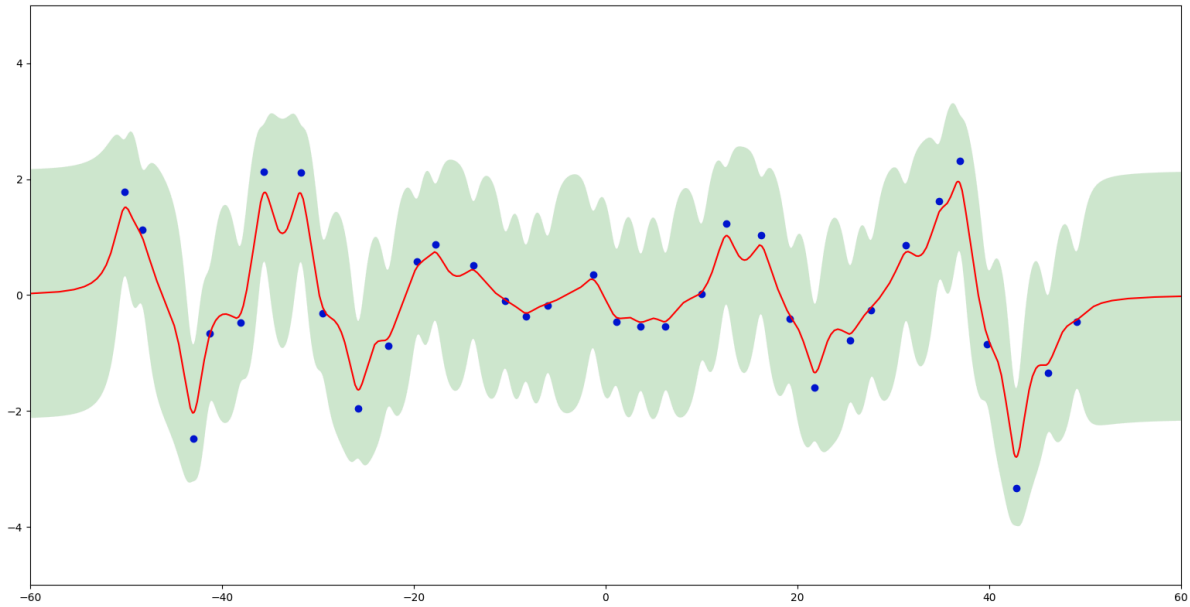
  - So, I design the **negative_log_likelihood_loss** function in **opt** method, and use *scipy.optimize.minimize* to minimize it.
  - The negative log likelihood is derived as $\frac{1}{2}\ln|C_{\alpha,l}| + \frac{1}{2}y^T C_{\alpha,l}^{-1} y + \frac{N}{2}\ln(2\pi)$, where N is the number of sample data points.
  - Then, I minimize it and update the hyper-parameters α and l in class.

```
def run(self):
    # predict with initial parameter setting (alpha = 1 and l = 1)
    self.predict()

    # minimize negative log likelihood and then predict
    self.opt()
    self.predict()
```

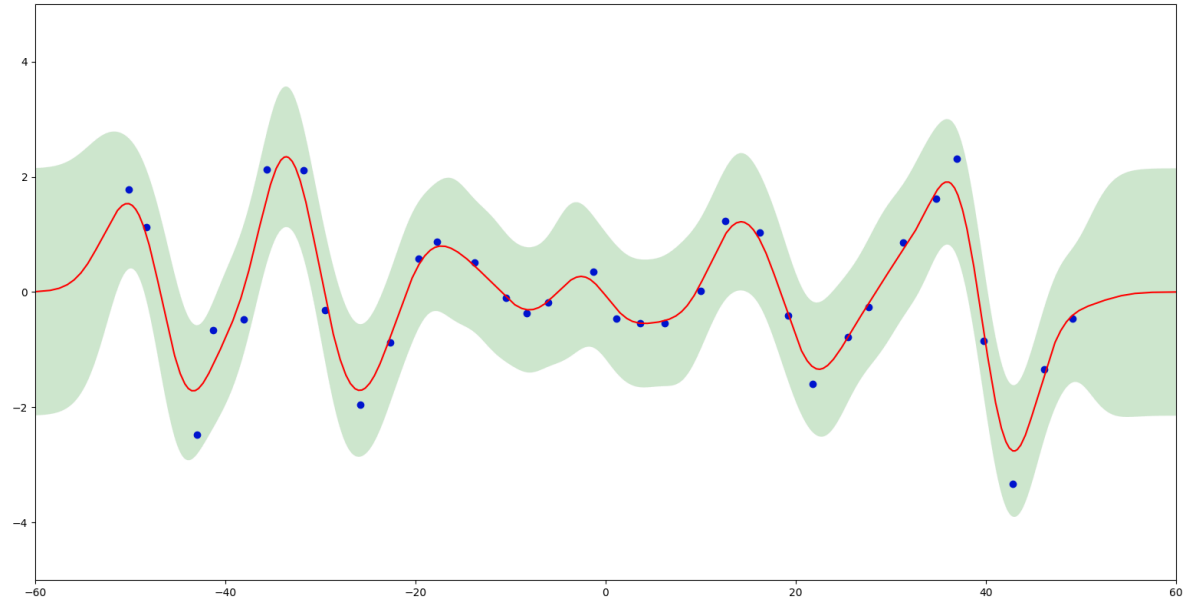
- Finally, I first run the **opt** method, and then run the **predict** method again to show the results for Task 2.

- **Experiment for task 1**



  - This is the simulation result for Task 1 ($\alpha = 1$ and $l = 1$).

- **Experiment for task 2**



  - This is the simulation result for Task 2 (updated $\alpha = 418.48$ and $l = 2.96$).

- **Observations and Discussion**
  - The updated $\alpha = 418.48$ and $l = 2.96$ (initial $\alpha = 1$ and $l = 1$).
  - I found that there are multiple local optimal solutions for α and l. For example, when I set initial $\alpha = 1$ and $l = 10$, the convergence is in $\alpha = 428.29$ and $l = 2.97$.
  - When initial $\alpha = 1$ and $l = 20$, the convergence is in $\alpha = 5336.26$ and $l = 2.96$, ... This indicates that choosing good initial parameters is crucial, but it remains a challenge!

SVM on MNIST

- **Code for task 1**

```
class SVM:
    def __init__(self, train_X, train_Y, test_X, test_Y):
        self.train_x = train_X
        self.train_y = train_Y
        self.test_x = test_X
        self.test_y = test_Y
        self.kernel_list = ['0', '1', '2']  # 0: Linear, 1: Polynomial, 2: RBF
```

  - First, I construct **SVM** class, which contains some *methods* and *variables* as follows:
    - *train_x*: the x-coordinates of training data points.
    - *train_y*: the y-coordinates of training data points.
    - *test_x*: the x-coordinates of testing data points.
    - *test_y*: the y-coordinates of testing data points.
    - *kernel_list*: the parameters for *libsvm*.

```
def run(self, task):
    if task == '1':    # task 1
        with open('SVM_task_1.txt', 'w') as f:
            for k in self.kernel_list:
                model = svm_train(self.train_y, self.train_x, '-t {}'.format(k))
                res = svm_predict(self.test_y, self.test_x, model)
                print("{}".format(res[1][0]), file=f)
```

  - In **run** method, we need to use (linear, polynomial, and RBF kernels) in svm, and compare their performance for Task 1. Therefore, I use the *svm_train* and *svm_predict* functions from *libsvm*, and set the parameter "-t " for *svm_train* to specify the kernel type (Reference (https://github.com/cjlin1/libsvm/blob/master/README)) as

follows.

```
`svm-train' Usage
================

Usage: svm-train [options] training_set_file [model_file]
options:
-s svm_type : set type of SVM (default 0)
        0 -- C-SVC              (multi-class classification)
        1 -- nu-SVC             (multi-class classification)
        2 -- one-class SVM
        3 -- epsilon-SVR        (regression)
        4 -- nu-SVR             (regression)
-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)
        3 -- sigmoid: tanh(gamma*u'*v + coef0)
        4 -- precomputed kernel (kernel values in training_set_file)
```

- Therefore, I set the parameters "-t 0", "-t 1", and "-t 2" in *svm_train* to generate the model (with default settings for other parameters). Then, I use *svm_predict* to make predictions and save the results to the file (SVM_task_1.txt).

- **Code for task 2**

```python
def run(self, task):
    if task == '1':    # task 1
        with open('SVM_task_1.txt', 'w') as f:
            for k in self.kernel_list:
                model = svm_train(self.train_y, self.train_x, '-t {}'.format(k))
                res = svm_predict(self.test_y, self.test_x, model)
                print("{}".format(res[1][0]), file=f)

    if task == '2':    # task 2
        self.grid_search()
```

- For Task 2, we need to use grid search to find the local best parameters for each kernel. So I implement the **grid_search** method to do this.

```python
def grid_search(self):
    costs = np.logspace(-5, 15, num=5, base=2, dtype=float) # cost: 2^-5 ~ 2^15
    gammas = np.logspace(-15, 3, num=5, base=2, dtype=float)    # gamma: 2^-15 ~ 2^3
    degs = [1, 3, 5]    # degree: 1, 3, 5
    coef0s = [0, 1]  # coef0: 0, 1

    opt_l_param, opt_p_param, opt_r_param = '', '', ''
    opt_l, opt_p, opt_r = 0, 0, 0
```

- In **grid_search** method, I aim to determine the optimal parameters for each kernel based on the following options:

- *cost*: $\{2^{-5}, 2^{0}, 2^{5}, 2^{10}, 2^{15}\}$ (for all kernels).
- *gamma*: $\{2^{-15}, 2^{-10.5}, 2^{-6}, 2^{-1.5}, 2^{3}\}$ (for polynomial and RBF kernels).
- *degree*: $\{1, 3, 5\}$ (for polynomial kernel).
- *coef0*: $\{0, 1\}$ (for polynomial kernel).

```
-d degree : set degree in kernel function (default 3)
-g gamma : set gamma in kernel function (default 1/num_features)
-r coef0 : set coef0 in kernel function (default 0)
-c cost : set the parameter C of C-SVC, epsilon-SVR, and nu-SVR (default 1)
-n nu : set the parameter nu of nu-SVC, one-class SVM, and nu-SVR (default 0.5)
-p epsilon : set the epsilon in loss function of epsilon-SVR (default 0.1)
-m cachesize : set cache memory size in MB (default 100)
-e epsilon : set tolerance of termination criterion (default 0.001)
-h shrinking : whether to use the shrinking heuristics, 0 or 1 (default 1)
-b probability_estimates : whether to train a model for probability estimates, 0 or 1 (default 0)
-wi weight : set the parameter C of class i to weight*C, for C-SVC (default 1)
-v n: n-fold cross validation mode
-q : quiet mode (no outputs)
```

```
for k in self.kernel_list:
    for c in costs:
        if k == '0':    # linear
            param = '-t {} -v 3 -c {:f}'.format(k, c)
            res = svm_train(self.train_y, self.train_x, param)
            if res > opt_1:
                opt_1 = res
                opt_1_param = param
```

- First, we run the *svm_train* with different cost values for linear kernel by setting the parameter "-t 0 -v 3 -c cost".
- Note that "-v " is for cross validation mode, and *svm_train* will return the accuracy.

```
-d degree : set degree in kernel function (default 3)
-g gamma : set gamma in kernel function (default 1/num_features)
-r coef0 : set coef0 in kernel function (default 0)
-c cost : set the parameter C of C-SVC, epsilon-SVR, and nu-SVR (default 1)
-n nu : set the parameter nu of nu-SVC, one-class SVM, and nu-SVR (default 0.5)
-p epsilon : set the epsilon in loss function of epsilon-SVR (default 0.1)
-m cachesize : set cache memory size in MB (default 100)
-e epsilon : set tolerance of termination criterion (default 0.001)
-h shrinking : whether to use the shrinking heuristics, 0 or 1 (default 1)
-b probability_estimates : whether to train a model for probability estimates, 0 or 1 (default 0)
-wi weight : set the parameter C of class i to weight*C, for C-SVC (default 1)
-v n: n-fold cross validation mode
-q : quiet mode (no outputs)
```

- If the new setting parameter is better than the current best one, we update and record it.

```
if k == '1':    # polynomial
    for g in gammas:
        for d in degs:
            for co in coef0s:
                param = '-t {} -v 3 -c {:f} -g {:f} -d {:d} -r {:d}'.format(k, c, g, d, co)
                res = svm_train(self.train_y, self.train_x, param)
                if res > opt_p:
                    opt_p = res
                    opt_p_param = param
```

- Second, the process is similar, but we need to add additional parameters to tune for the polynomial kernel "-t 1 -v 3 -c cost -g gamma -d degree -r coef0".
- If the new setting parameter ... (same as the front)

```
if k == '2':    # RBF
    for g in gammas:
        param = '-t {} -v 3 -c {:f} -g {:f}'.format(k, c, g)
        res = svm_train(self.train_y, self.train_x, param)
        if res > opt_r:
            opt_r = res
            opt_r_param = param
```

- Finally, we follow the similiar step to tune for the RBF kernel "-t 2 -v 3 -c cost -g gamma".
- If the new setting parameter ... (same as the front)

- **Code for task 3**
  - For Task 3, we need to design the user-defined kernel (linear+RBF).

```
def linear_kernel(self, x1, x2):
    return np.dot(x1, x2.T)

def RBF_kernel(self, x1, x2, gamma):
    dis = np.sum(x1**2, 1).reshape(-1, 1) + np.sum(x2**2, 1) - 2 * np.dot(x1, x2.T)
    return np.exp(-gamma * dis)

def my_kernel(self):
    gamma = 0.00127
    train_kernel = self.linear_kernel(self.train_x, self.train_x) \
                   + self.RBF_kernel(self.train_x, self.train_x, gamma)
    test_kernel = self.linear_kernel(self.test_x, self.train_x) \
                  + self.RBF_kernel(self.test_x, self.train_x, gamma)
    train_kernel = np.hstack((np.arange(1, len(self.train_x) + 1).reshape(-1, 1), train_kernel))
    test_kernel = np.hstack((np.arange(1, len(self.test_x) + 1).reshape(-1, 1), test_kernel))
    return train_kernel, test_kernel
```

  - We need to precompute the kernel and make it match the input format in *svm_train* as shown below:

```
Precomputed Kernels
==================

Users may precompute kernel values and input them as training and
testing files.  Then libsvm does not need the original
training/testing sets.

Assume there are L training instances x1, ..., xL and.
Let K(x, y) be the kernel
value of two instances x and y. The input formats
are:

New training instance for xi:

<label> 0:i 1:K(xi,x1) ... L:K(xi,xL)

New testing instance for any x:

<label> 0:? 1:K(x,x1) ... L:K(x,xL)

That is, in the training file the first column must be the "ID" of
xi. In testing, ? can be any value.
```

- So, in **my_kernel** method, I first construct the train kernel and test kernel (linear kernel + RBF kernel). Note that I set $\gamma = 0.00127$ (default) in RBF kernel.
- Then, I add the index for them to match the format.

```python
def run(self, task):
    if task == '1':    # task 1
        with open('SVM_task_1.txt', 'w') as f:
            for k in self.kernel_list:
                model = svm_train(self.train_y, self.train_x, '-t {}'.format(k))
                res = svm_predict(self.test_y, self.test_x, model)
                print("{}".format(res[1][0]), file=f)

    if task == '2':    # task 2
        self.grid_search()

    if task == '3':    # task 3
        train_kernel, test_kernel = self.my_kernel()
        model = svm_train(self.train_y, train_kernel, '-t 4')
        res = svm_predict(self.test_y, test_kernel, model)
        with open('SVM_task_3.txt', 'w') as f:
            print("{}".format(res[1][0]), file=f)
```

- Then, we use the precomputed kernel to train and predict, and record the results.
- Note that the parameter for *svm_train* is set to "-t 4", since we use the precomputed kernel.

```
-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)
        3 -- sigmoid: tanh(gamma*u'*v + coef0)
        4 -- precomputed kernel (kernel values in training_set_file)
```

- **Experiment for task 1**

```
95.08
34.68
95.32000000000001
```

  - The accuracy in different kernels (with default parameter settings):
    - Linear: 0.9508
    - Polynomial: 0.3468
    - RBF: 0.9532

- **Experiment for task 2**

```
kernel: {Linear}, OPT = {96.860000}, OPT_params = {-t 0 -v 3 -c 0.031250}
kernel: {Polynomial}, OPT = {97.980000}, OPT_params = {-t 1 -v 3 -c 1.000000 -g 0.015625 -d 3 -r 1}
kernel: {RBF}, OPT = {98.420000}, OPT_params = {-t 2 -v 3 -c 32.000000 -g 0.015625}
```

  - By grid search, we find the *best* performance and corresponding parameters for each kernel as follows:
    - Linear:
      - accuracy: 0.9686
      - cost: 0.03125
    - Polynomial:
      - accuracy: 0.9798
      - cost: 1
      - gamma: 0.015625
      - degree: 3
      - coef0: 1
    - RBF:
      - accuracy: 0.9842
      - cost: 32
      - gamma: 0.015625

- **Experiment for task 3**
  - The accuracy for linear kernel + RBF kernel is 0.9508 (with default parameter settings).

- **Observations and Discussion**
  - I observe that the performance of Task 3 and linear kernel in Task 1 is similiar and worse than RBF kernel in Task 1. Therefore, I try to adjust the parameter γ in RBF kernel, and observe the performance of Task 3 as follows:
    - $\gamma = 0.0001$, accuracy = 0.9508
    - $\gamma = \frac{1}{\#\text{feature}} = 0.00127$ (default), accuracy = 0.9508
    - $\gamma = 0.01$, accuracy = 0.9532
    - $\gamma = 0.05$, accuracy = 0.9568
    - $\gamma = 0.08$, accuracy = 0.9564
  - Thus, I think the choice of γ (and other hyper-parameters) is critical for balancing the model's complexity and its adaptation to the data. Further tuning of γ or other parameters may help achieve optimal performance for Task 3.