Machine Learning Homework 5

311554046 林愉修

I. Gaussain Process

a. code with detailed explanations

Part 1.

- Load Data I defined a load_data function in dataloader.py.

```
# dataloader.py
def load_data(data_path):
    X = []
    y = []
    with open(os.path.join(data_path, 'input.data'), 'r') as f:
        for line in f.readlines():
            line = line.split()
            X.append(float(line[0]))
            y.append(float(line[1]))

    return np.array(X), np.array(y)
```

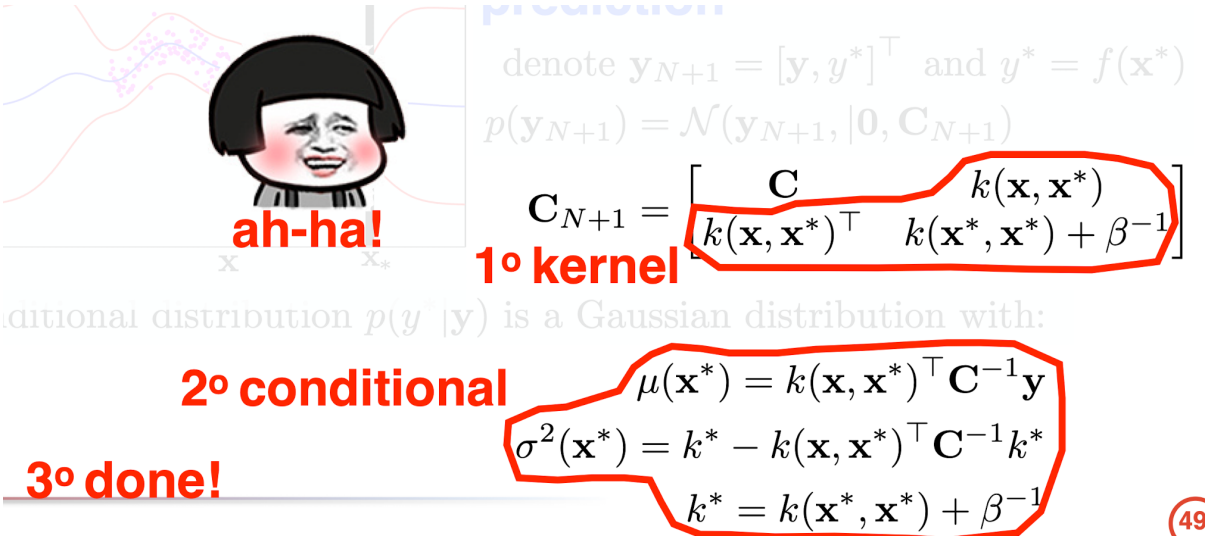
- Rational Quadratic Kernel
  Follow the formula referenced from peterroelants.github.io, I defined a rational_quadratic_kernel function in kernel.py.

$$k(x_a, x_b) = \sigma^2 \left(1 + \frac{||x_a - x_b||^2}{2\alpha\ell^2}\right)^{-\alpha}$$

```
# kernel.py
def rational_quadratic_kernel(x_a, x_b, **kernel_param):
    sigma = kernel_param.get('sigma', 1.0)
    length_scale = kernel_param.get('length_scale', 1.0)
    alpha = kernel_param.get('alpha', 1.0)
    SE = np.power(x_a.reshape(-1, 1) - x_b.reshape(1, -1), 2)

    return sigma**2 * np.power(1 + SE / (2 * alpha * length_scale**2), -
alpha)
```

- Gaussian Process



I followed the three steps in the slides, and declare a class called GaussianProcess in GaussianProcess.py, you can choose whatever kernel you want as well as the kerenl parameter for the GaussianProcess.

```
# GaussianProcess.py
class GaussianProcess:
    def __init__(self, kernel_func, *, beta=5, **kernel_param):
        self.kernel_func = kernel_func
        self.kernel_param = kernel_param
        self.beta = beta
        self.X = None
        self.y = None
        self.C = None
        self.x_star = None
        self.mean = None
        self.var = None
        self.std = None
```

So the first step is to compute the covariance C, following formula in the slides, I defined a private member function called _covariance for the GaussianProcess class.

```
# GaussianProcess.py
def _covariance(self, X):
    return self.kernel_func(X, X, **self.kernel_param) + 1 / self.beta *
np.identity(len(X))
```

Moreover, I defined a public member function called fit for the GaussianProcess class, which basically is computing the covariance for training data X.

```
# GaussianProcess.py
def fit(self, X):
    self.X = X
    self.C = self._covariance(self.X)
```

```
    return self
```

Finally, I combined the three step meteioned above, calculating all the stuffs in the public member function called predict for the GaussianProcess class.

```
# GaussianProcess.py
def predict(self, x_star, y):
    self.y = y
    self.x_star = x_star
    k_x_s = self.kernel_func(self.X, x_star, **self.kernel_param)
    k_star = self.kernel_func(x_star, x_star, **self.kernel_param) + 1 /
self.beta * np.identity(len(x_star))
    temp = k_x_s.T @ np.linalg.inv(self.C)
    self.mean = temp @ self.y
    self.var = k_star - temp @ k_x_s
    self.std = np.sqrt(np.diag(self.var))

    return self.mean, self.var
```

- Visualization
  And for the visualization, I defined a public member function called visualization for the Gaussain Process class.

```
# GaussianProcess.py
def visualization(self, title, fig_path, fig_name='figure.png'):
    plt.figure(figsize=(15, 5))
    plt.plot(self.x_star, self.mean, color='lightseagreen', label='mean')
    plt.fill_between(self.x_star, self.mean + 2 * self.std, self.mean - 2 *
self.std, facecolor='aquamarine', label='95% confidence interval')
    plt.scatter(self.X, self.y, color='mediumvioletred', label='training
data')
    plt.legend(loc='upper right')
    plt.title(title)
    plt.savefig(os.path.join(fig_path, fig_name))
```

Part 2.

- Optimize the kernel parameters
  Here we tried to optimize the kernel parameters by minimizing the negative log likelihood, I used the scipy.optimize.minimize to do so, where I defined a public member function called optimize_kernel_param for the GaussianProcess class.

```
# GaussianProcess.py
def optimize_kernel_param(self, X, y, *, bounds=None, **init_param):
    self.X = X
```

```
    self.y = y
    self.kernel_param = init_param
    const_args = (self.X, self.y, self.beta)
    x0 = tuple(init_param.values())

    res = minimize(self._objFunc(const_args), x0, bounds=bounds)
    for idx, key in enumerate(self.kernel_param.keys()):
        self.kernel_param[key] = res.x[idx]

    return self
```

- Negative Log Likelihood I followed the formula given in the slides, and defined a private member function called _obj_func for the GaussianProcess class, which is a nested definition that wrapped the negLogLikelihood function inside.

- Given $\mathcal{D} = \{(\mathbf{x}_i, y_i)_{n=1}^N\} = (\mathbf{X}, \mathbf{y})$, the marginal likelihood is function of $\boldsymbol{\theta}$

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

$$\ln p(\mathbf{y}|\theta) = -\frac{1}{2}\ln |\mathbf{C}_\theta| - \frac{1}{2}\mathbf{y}^\top \mathbf{C}_\theta^{-1}\mathbf{y} - \frac{N}{2}\ln (2\pi) \quad \text{☞} \quad \frac{\partial \ln p(\mathbf{y}|\theta)}{\partial \theta}$$

```
# GaussianProcess.py
def _objFunc(self, const_args):
    X, y, beta = const_args

    def negLogLikeLihood(x0):
        for idx, key in enumerate(self.kernel_param.keys()):
            self.kernel_param[key] = x0[idx]
        self.C = self._covariance(X)
        return 0.5 * np.log(np.linalg.det(self.C)) + 0.5 * y.T @
np.linalg.inv(self.C) @ y + len(X) / 2 * np.log(2 * np.pi)

    return negLogLikeLihood
```

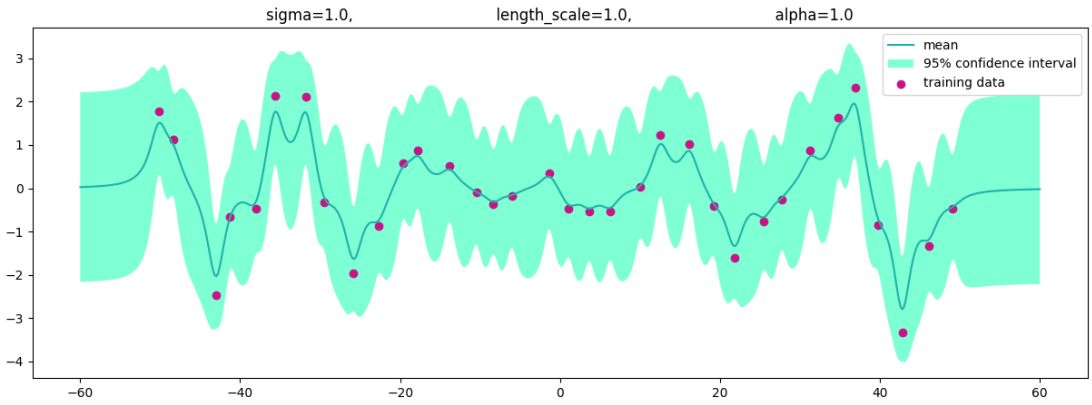
b. experiments settings and results

Part 1.

I use beta=5 and kernel parameters (sigma, length_scale, alpha) = (1, 1, 1).

```
# main.py
GP = GaussianProcess(rational_quadratic_kernel, beta=5)
mean, var = GP.fit(X_train).predict(X_test, y_train)
GP.visualization(title=f"sigma=1.0, \
                         length_scale=1.0, \
                         alpha=1.0",
```

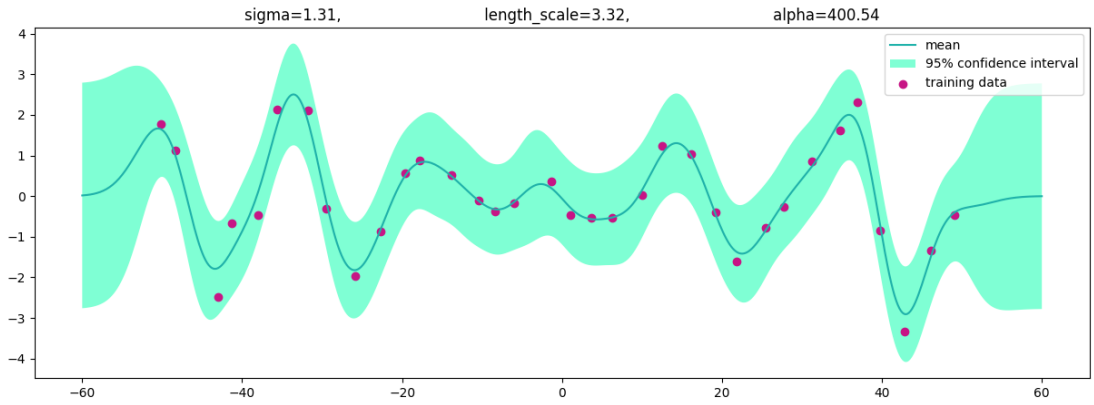
```
                    fig_path=args.fig_path,
                    fig_name='original.png')
```



Part 2.

I use beta=5, and set the initial kernel parameters (sigma, length_scale, alpha) = (1, 1, 1), and found the optimal kernel parameters (1.31, 3.32, 400.54).

```
# main.py
mean_opt, var_opt = GP.optimize_kernel_param(
                            X_train,
                            y_train,
                            bounds=((1e-6, None),
                                    (1e-6, None),
                                    (1e-6, None)),
                            sigma=1,
                            length_scale=1,
                            alpha=1) \
                        .predict(X_test, y_train)
GP.visualization(
        title=f"sigma={GP.kernel_param['sigma']:.2f}, \
                length_scale={GP.kernel_param['length_scale']:.2f}, \
                alpha={GP.kernel_param['alpha']:.2f}",
        fig_path=args.fig_path,
        fig_name='optimized.png')
```



c. observations and discussion

As the results shown in section b, by using the optimized kernel parameter, comparing to default kernel parameters, the predicted mean is less peaky and the predicted variance is smaller on every datapoint, thus the predicted line looks more smoother.

II. SVM

a. code with detailed explanations

Part 1.

- Load Data
  I used the pandas.read_csv to load all the data and transform them to numpy array.

```
# dataloader.py
def load_data(data_path):
    X_train = pd.read_csv(os.path.join(data_path, "X_train.csv"),
header=None).to_numpy()
    y_train = pd.read_csv(os.path.join(data_path, "Y_train.csv"),
header=None).to_numpy().reshape(-1)
    X_test = pd.read_csv(os.path.join(data_path, "X_test.csv"),
header=None).to_numpy()
    y_test = pd.read_csv(os.path.join(data_path, "Y_test.csv"),
header=None).to_numpy().reshape(-1)

    return X_train, y_train, X_test, y_test
```

- Using different kernel
  I follow the libsvm svm_train usage reference from its libsvm github repo and libsvm/python

```
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)
-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)
```

I defined a dictionary called kernel_type to map a specific kernel type to its corresponding argument value.

```python
# main.py
kernel_type = {"linear": 0,
               "polynomial": 1,
               "RBF": 2,
               "sigmoid": 3,
               "precomputed": 4}
```

By passing the kernel argument in svm_train function, I got the result for a given kernel type.

```python
# main.py
for key, value in kernel_type.items():
    if key == 'precomputed' or key == 'sigmoid':
        continue
    m = svm_train(y_train, X_train, f"-q -t {value}")
    p_labels, p_acc, p_vals = svm_predict(y_test, X_test, m, "-q")
    print(f"kernel_type: {key}\ttesting accuracy: {p_acc[0]:.2f}", file=f)
```

Part 2.

- Use C-SVC
  As the libsvm usage shown above, the default type of SVM is already C-SVC, so I didn't explicitly pass the -s argument value.

- Grid Search
  Here, I defined a function called gridSearch, which you can set kerenl type and the kernel parameters (and the cost for C-SVC, of course), if one is not given, it will use the default value. The parameters is given in a list, which is the seach space for the parameter, I used itertools.product to get all the combinations of the parameters value for searching, and choose the combination with the highest accuracy in the 3-fold cross validation.

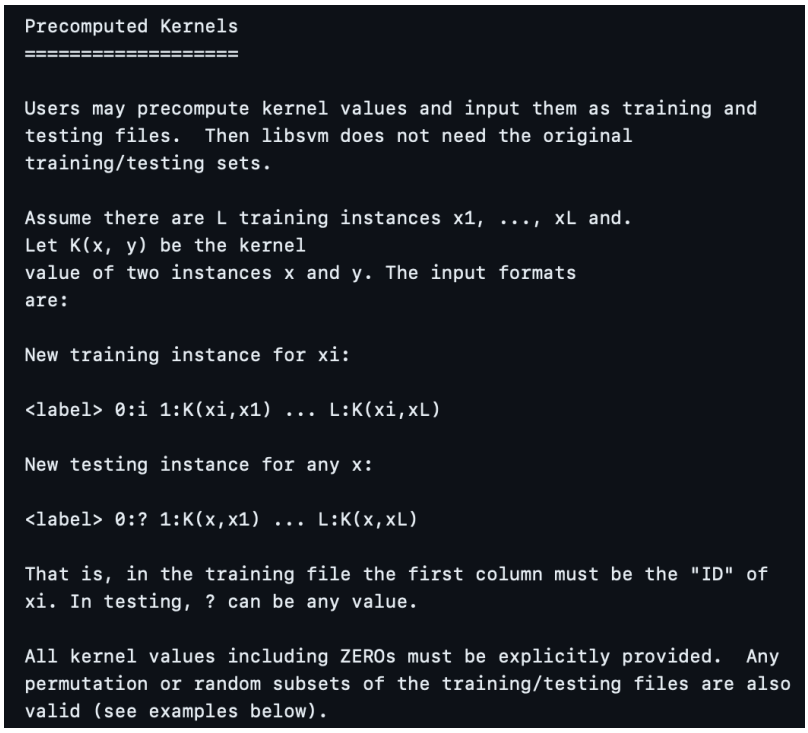
```
# gridSearch.py
def gridSearch(X_train, y_train, **param):
    kernel_type = param.get("kernel_type", 0)
    C = param.get("C", [1])
    gamma = param.get("gamma", [1 / X_train.shape[1]])
    coef0 = param.get("coef0", [0])
    degree = param.get("degree", [3])

    combinations = [C, gamma, coef0, degree]
    best_acc = 0
    best_comb = None
    for comb in list(itertools.product(*combinations)):
        acc = svm_train(y_train, X_train, f"-q -t {kernel_type} -v 3 -c
{comb[0]} -g {comb[1]} -r {comb[2]} -d {comb[3]}")
        if acc > best_acc:
            best_acc = acc
            best_comb = comb

    print(f"best combination (C, gamma, coef0, degree): {best_comb}\tbest
accuracy: {best_acc}")
    return best_comb, best_acc
```

Part 3.

- Using linear + RBF kernel
  I followed the instruction referenced from libsvm github repo and stackoverflow.com and defined a new kernel that combined linear kernel and RBF kernel.
  And for the RBF kernel, I used scipy.spatial.distance.cdist to calculate the squared euclidean distance between two x.



```
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.

All kernel values including ZEROs must be explicitly provided.  Any
permutation or random subsets of the training/testing files are also
valid (see examples below).
```

```python
# precomputed_kernel.py
def linearRBF(X, X_, gamma):
    linear = X @ X_.T
    RBF = np.exp(-gamma * cdist(X, X_, 'sqeuclidean'))
    kernel = linear + RBF
    kernel = np.hstack((np.arange(1, len(X)+1).reshape(-1, 1), kernel))

    return kernel
```

Using isKernel=True in svm_problem function to use the precomputed kernel.

```python
# main.py
K = linearRBF(X_train, X_train, best_comb[1])
KK = linearRBF(X_test, X_train, best_comb[1])
prob = svm_problem(y_train, K, isKernel=True)
m = svm_train(prob, f"-q -t {kernel_type['precomputed']} -c
{best_comb[0]}")
p_labels, p_acc, p_vals = svm_predict(y_test, KK, m, "-q")
print(f"kernel_type: linear + RBF kernel\ttesting accuracy:
{p_acc[0]:.2f}", file=f)
```

b. experiments settings and results

Part 1.

result:

```
kernel_type: linear testing accuracy: 95.08
kernel_type: polynomial testing accuracy: 34.68
kernel_type: RBF    testing accuracy: 95.32
```

Part 2.

- Linear kernel

search space:

```
param = {"kernel_type": kernel_type['linear'],
         "C": [10**x for x in range(-5, 6)]}
```

result:

```
best combination (C): (0.01)
best training accuracy: 96.90
after grid search testing accuracy: 95.96
```

- Polynomial kernel

search space:

```
param = {"kernel_type": kernel_type['polynomial'],
         "C": [10**x for x in range(-3, 4)],
         "gamma": [10**x for x in range(-3, 4)],
         "coef0": [x for x in range(-1, 2)],
         "degree": [x for x in range(2, 5)]}
```

result:

```
best combination (C, gamma, coef0, degree): (1, 1, 1, 2)
best training accuracy: 98.16
after grid search testing accuracy: 97.72
```

- RBF kernel

search space:

```
param = {"kernel_type": kernel_type['RBF'],
         "C": [10**x for x in range(-3, 4)],
         "gamma": [10**x for x in range(-3, 4)]}
```

result:

```
best combination (C, gamma): (100, 0.01)
best training accuracy: 98.34
after grid search testing accuracy: 97.52
```

Part 3.

I set (C, gamma) from RBF kernel grid search result, which is (100, 0.01).
result:

```
kernel_type: linear + RBF kernel    testing accuracy: 95.32
```

c. observations and discussion

As the results shown in section b, the testing accuracy of C-SVC with polynomial kernel increases the most after grid search, and there are three parameter that has changed, one is gamma, it changes from $\frac{1}{784}$ to 1 another is coef0, it changes from 0 to 1, and the third is degree, it changes from 3 to 2.
And I found that if you only increase C, the accuracy would increase a lot. If you only increase gamma, the accuracy would also increase a lot. If you only choose the value of coef0 signicantly deviate from 0, the accuracy would also increse a lot.
If you only set the value of degree smaller than 3, the accuracy would also increase a lot.