ML HW5 Report

0510008 藍挺毓

I. Gaussian Process

A. Code

1. There are two parts in Gaussian Process, which is optimize kernel parameters or not. I use argument —optimize to determine whether optimize parameters in rational quadratic kernel.

```
parser = ArgumentParser()
parser.add_argument("--optimize", type=int, default=1)
args = parser.parse_args()
```

2. Kernel: I defined rational quadratic function according to the definition.

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

```
var * (1+cdist(xa, xb, 'sqeuclidean')/2/alpha/l/l)**(-alpha)
```

3. Gaussian Process: According to the lecture, C is a set of kernels that represents features' distance between all training data. kxx represents kernel distance between training and testing data. k* is kernel distance between all testing data with some random noise. Beta means how difference noise is.
Then, I calculate mean and variance according to the following equations.

$$\mu(\mathbf{x}^*) = k(\mathbf{x}, \mathbf{x}^*)^\top \mathbf{C}^{-1}\mathbf{y}$$
$$\sigma^2(\mathbf{x}^*) = k^* - k(\mathbf{x}, \mathbf{x}^*)^\top \mathbf{C}^{-1} k(\mathbf{x}, \mathbf{x}^*)$$
$$k^* = k(\mathbf{x}^*, \mathbf{x}^*) + \beta^{-1}$$

```
C = RationalQuadraticKernel(X, X, alpha, l)
kxx = RationalQuadraticKernel(X, x_test, alpha, l)
kstar = np.add(RationalQuadraticKernel(
    x_test, x_test, alpha, l), np.eye(len(x_test))*1.0/beta)

mu = kxx.T.dot(np.linalg.inv(C)).dot(Y)  # 500,1
var = kstar - kxx.T.dot(np.linalg.inv(C)).dot(kxx)  # 500,500
```

4. Z score equals to 1.96 when we have 95 percent confidence on prediction. Therefore, I times variance with 1.96 z score, and drew it with transparent blue on my figure.

```
for i in range(points):
    upper[i] = mu[i, 0] + var[i, i]*1.96
    lower[i] = mu[i, 0] - var[i, i]*1.96

visualize(mu, upper, lower)
```

```
def visualize(mu, upper, lower):
    plt.xlim(-60, 60)
    plt.scatter(X, Y, c='r', edgecolors='face')
    plt.plot(x_test.ravel(), mu.ravel(), 'b')
    plt.fill_between(x_test.ravel(), upper, lower, alpha=0.3)
    plt.show()
```

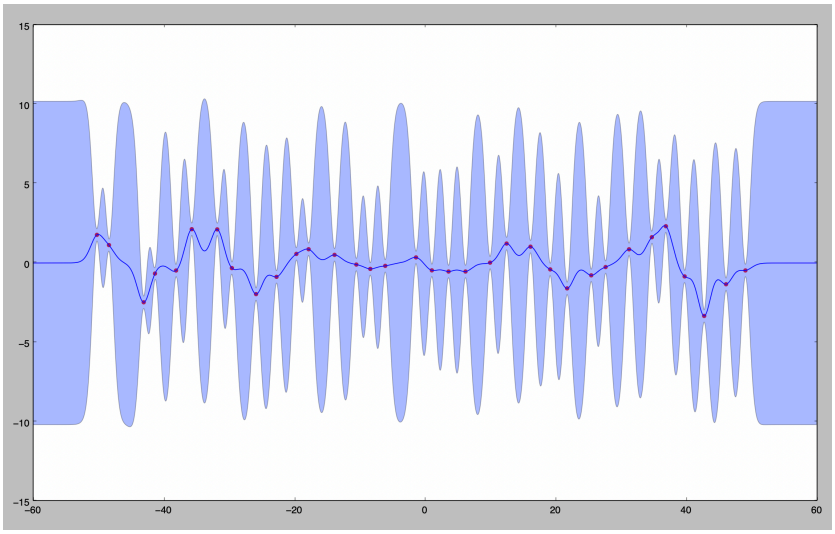
5. Optimize: To optimize parameters in kenel. I call optimize.minimize function to find best parameters. I first derive kernel's log likelihood function. Then, call built-in optimize function. After finding best parameters, just call same Gaussian Process function with updated parameters.

```
def marginal_likelihood(theta):
    C = RationalQuadraticKernel(X, X, alpha=theta[0], l=theta[1])
    return 0.5*Y.T.dot(np.linalg.inv(C)).dot(Y) + 0.5*math.log(np.linalg.det(C)) + n/2*math.log(2.0*math.pi)
```

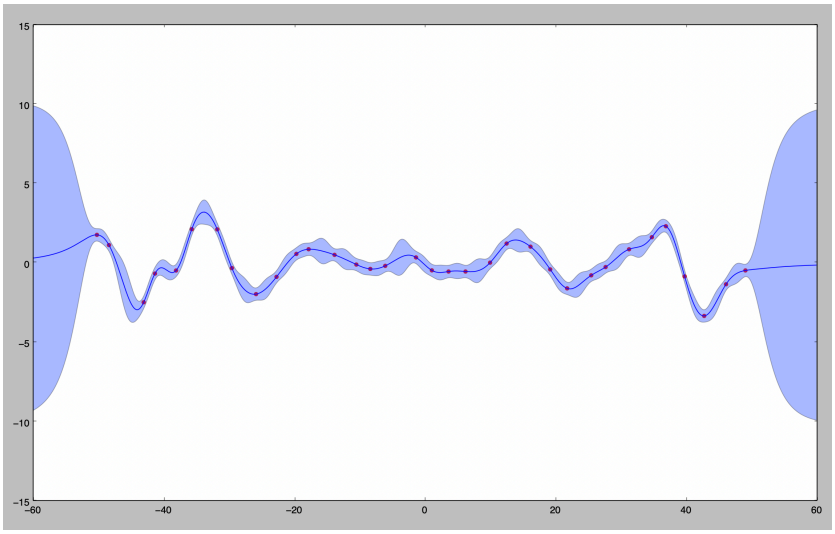
```
theta = [1.0, 1.0]
res = minimize(marginal_likelihood, theta)
```

B. Results

1. Random picked parameters.
   alpha=10, l=1



2. Optimized parameters by minimizing negative marginal log-likelihood.



C. Discussion

1. result 2 is obviously better than result 1.
2. We can see from the result that with training data point, gaussian process has a higher confidence with prediction it made. Between training data, gaussian process may still make fine guess. However, as to interval that there is none of the training data, gaussian process can hardly make any prediction. As a result, we have out 95% confidence region much larger than other parts.
3. Although there is a variance in rational quadratic function, it is not important. Every kernel has this parameter out in front, and it is just a scale factor. Variance means how far the function away from its mean.
4. It may show really bad result if we have bad parameters. The worst case of overfitting is multiple impulse function. It fit training data well. However, it is obviously that it is not the general function we are trying to find.

II. SVM

A. Code

1. To run different part of the assignment, user should change ‘mode’.
2. I first read MNIST as my dataset

```
def read_dataset(filename):
    with open(filename,'r') as fp:
        lines = fp.readlines()
    lines = np.array( [line.strip().split(',') for line in lines] , dtype="float64")
    return lines

def read_mnist():
    X_train  = read_dataset("X_train.csv")  #(5000,784)
    X_test = read_dataset("X_test.csv")  #(2500,784)
    Y_train = read_dataset("Y_train.csv")  #(5000,1)
    Y_test = read_dataset("Y_test.csv")  #(2500,1)

    Y_train = Y_train[:,0]
    Y_test = Y_test[:,0]

    return X_train, X_test, Y_train, Y_test
```

3. Part A: Simply calls built-in function with default parameters.
   -q: not to print while training
   -t: choose kernel functions. 0,1,2 means linear, polynomial, RBF separately.
   svm_predict return three terms, which are predicted labels, accuracy, and val.
   Accuracy include three different values: accuracy, mean-square error, snd squared correlation coefficient.

```
def svm(kernel_type, param_str, record_time=False): #-v n: n-fold vross validation mode
    print("\n", kernel_type)
    param_str = param_str + " -q" #-s svm_type: default=0=C_SVC
    param = svm_parameter(param_str)

    time_start = time.time()
    prob = svm_problem(Y_train, X_train)
    model = svm_train(prob, param)
# ******* part A *******                                    uared correlation coefficient(for regression)
if mode==0:                                                 l, p_acc, p_val
    svm("Linear", "-t 0")  #-c
    svm("Polynomial", "-t 1")  #-c, -r, -g, -d  tart) )
    svm("RBF", "-t 2")  #-c, -g
```

My self-defined svm function would train and predict according to input kernel type.

5. Part B: I make a self-defined GridSearch Function to deal with whole grid search process.
   Within GridSearch(), I first defined some parameters. The whole grid search process went through all these parameters and find the best one.
   For each kernel type, my code simply walked through all defined parameters and do cross validation. Also, record best accuracy and parameters while waling through all parameters.
   • some new parameters:
   -c cost: set C of C-SVC
   -d degree: set degree in kernel function default is 3
   -g gamma: default is 1/num_features
   -r coef0: default is 0
   -v n: n-fold cross validation, without using -v means not using cross validation

```
# ******* part B *******
if mode==1:
    GridSearch("Linear")
    GridSearch("Polynomial")
    GridSearch("RBF")
```

```
def GridSearch(kernel_type):
    c = [1e-4, 1e-3, 1e-2, 0.1, 1, 10, 100, 1000, 10000]
    d = range(0, 10)
    g = [1/784, 1, 1e-1, 1e-2, 1e-3, 1e-4]
    r = range(-10, 10, 1)
    n = 3

    acc_record = []
    max_acc = 0.0
    if kernel_type == "Linear":
        for ci in range(len(c)):
            param_str = " -c "+str(c[ci])
            acc = gridsearch_svm(kernel_type+param_str,
                                 "-t 0"+param_str, fold=n)
            # acc_record.append( acc )
            if acc > max_acc:
                max_acc = acc
                max_param = param_str
```

```
    elif kernel_type == "Polynomial":
        for ci in range(len(c)):
            param_str = " -c "+str(c[ci])
            for ri in range(len(r)):
                param_str_r = param_str+" -r "+str(r[ri])
                for gi in range(len(g)):
                    param_str_g = param_str_r+" -g "+str(g[gi])
                    for di in range(len(d)):
                        param_str_d = param_str_g+" -d "+str(d[di])
                        acc = gridsearch_svm(
                            kernel_type+param_str_d, "-t 1"+param_str_d, fold=n)
                        # acc_record.append( acc )
                        if acc > max_acc:
                            max_acc = acc
                            max_param = param_str_d

    elif kernel_type == "RBF":
        for ci in range(len(c)):
            param_str = " -c "+str(c[ci])
            for gi in range(len(g)):
                param_str_g = param_str+" -g "+str(g[gi])
                acc = gridsearch_svm(
                    kernel_type+param_str_g, "-t 2"+param_str_g, fold=n)
                # acc_record.append( acc )
                if acc > max_acc:
                    max_acc = acc
                    max_param = param_str_g
```

6. Part C: Self defined kernel function, linear+RBF.
   Libsvm supports user-defined kernel. Instead of input training data directly, we need to input well-calculated data in kernel space and set isKernel=True. Thus, I first calculated all training data into kernel space, then following similar svm process. I have done the

```
def linear_kernel(u, v):
    return u.dot(v.T)


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

```python
# ******* part C *******
if mode == 2:
    g = [1/784, 1, 1e-1, 1e-2, 1e-3, 1e-4]
    c = [1e-4, 1e-3, 1e-2, 0.1, 1, 10, 100, 1000, 10000]

    x = read_dataset("X_train.csv")  # (5000,784)
    row, col = x.shape

    max_acc = 0.0
    g_best = 0
    linear_k = linear_kernel(x, x)
    for gi in range(len(g)):
        rbf_k = RBF_kernel(x, x, -g[gi])
        my_k = linear_k + rbf_k
        my_k = np.hstack((np.arange(1, row+1).reshape(-1, 1), my_k))
        prob = svm_problem(Y_train, my_k, isKernel=True)
        for ci in range(len(c)):
            param_str = "-t 4 -c " + str(c[ci]) + " -v 3 -q"
            param_rec = "-t 4 -c " + str(c[ci]) + " -q"
            print("-g", g[gi], param_str)
            param = svm_parameter(param_str)
            val_acc = svm_train(prob, param)

            if val_acc > max_acc:
                max_acc = val_acc
                max_param = param_rec
                g_best = gi
    print("============================")
    print("Best Parameters:", " -g", g[g_best], max_param)
    print("Max accuracy:", max_acc)

    rbf_k = RBF_kernel(x, x, -g[g_best])
    my_k = linear_k + rbf_k
    my_k = np.hstack((np.arange(1, row+1).reshape(-1, 1), my_k))
    prob = svm_problem(Y_train, my_k, isKernel=True)
    param = svm_parameter(max_param)
    model = svm_train(prob, param)

    x_test = read_dataset("X_test.csv")  # (2500,784)
    row, col = x_test.shape
    linear_k = linear_kernel(x_test, x_test)
    rbf_k = RBF_kernel(x_test, x_test, -g[g_best])
    my_k = linear_k + rbf_k
    my_k = np.hstack((np.arange(1, row+1).reshape(-1, 1), my_k))
    _, test_acc, _ = svm_predict(Y_test, my_k, model)
    print("============================")
```

grid search, as well as train whole training data and predict testing data with best parameters fund in grid search.

B. Results

Part A			
Kernel		Testing Accuracy (%)	default parameters
Linear		95.08	
Polynomial		34.68	-g 1/784 -r 0 -d 3
RBF		95.32	-g 1/784
Part B - Grid Search with 3-fold cross-validation			
Kernel	Cross-validation Accuracy (%)	Testing Accuracy (%)	Parameters
Linear	96.24	96.04	-c 3
Polynomial	97.72	43.76	-c 0.01 -r 5 -g 1 -d 3
RBF	98.24	98.2	-c 10 -g 0.01
Part C - Linear+RBF			
Kernel	Cross-validation Accuracy (%)	Testing Accuracy (%)	
Linear+RBF	96.94	34.92	-g 0.001 -t 4 -c 0.1

C. Discussion

1. It is reasonable RBF has better performance than all the others. RBF makes everything linearly separable since it can map data into infinite dimension space. Thus, though MNIST has 784 feature space, and looks complicate. RBF allows svm to make good

classification. However, the accuracy is not 100 here. This is due to the reason that we only make limited search among parameters.

2. From the result, we verify RBF is a great kernel in classification. This is why people usually pick RBF as their kernel function.
3. It is reasonable to have better result in part B than in part A. Since B is doing same svm as A, but with best parameters it found in grid search.
4. From experiment A and B, we can see the importance of finding good parameters for different dataset.
5. Testing result may be worse than cross-validation result. Since best parameters found with training data may not be the best parameters for testing data. But the slice drop in accuracy is reasonable.
6. Linear kernel: There is not specific parameters need to be fine-tuned.
7. Polynomial kernel: Most time consuming while doing search since it need to fine-tuned a lots of parameters.
8. RBF kernel: a well-used kernel function for its great ability in classification.
9. C and gamma are two most important parameters to fine-tuned. C is the cost. The higher c value is, the less tolerance the model has to error. This results in too fit to training data. As to gamma, it determine the space RBF can project to. If gamma is smaller, it can support more vectors.
10. I wrote a timmer in my code. The result shows linear is the fasters among all kernel function. It is reasonable since linear kernel is the most simplest one.