0845034
LEMASQUERIER Max

Report

I – Gaussian Process

a) Code

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import minimize
from scipy.spatial.distance import *
```

Library used

```
# Read the input file
data = open('input.data','r').read().splitlines()
for i in range(len(data)):
    data[i] = data[i].split()
    data[i][0] = float(data[i][0])
    data[i][1] = float(data[i][1])
data = np.asarray(data)

```

This code load the input.data file into an array variable named data.

```
# Def kernel function
def kernel(x1,x2,param=[1,1,1]):
    """
        rational quadratic kernel, 3 parameters : sigma,alpha,l
    """
    l,sigma,alpha = param
    temp = cdist(x1,x2,'euclidean')
    return sigma**2*(1+temp/(2*alpha*l**2))**(-alpha)
```

Define the kernel function, here it's the rational quadratic kernel. It takes 3 parameter.

```python
def posterior_predictive(X_s, X_train, Y_train, param):
    '''
        Compute posterior from known parameters
    '''
    cov = kernel(X_train, X_train, param) + 1e-8 * np.eye(len(X_train))
    cov_train = kernel(X_train, X_s, param)
    K = kernel(X_s, X_s, param) + 1e-8 * np.eye(len(X_s))
    K_inv = np.linalg.inv(cov)

    mu_s = cov_train.T.dot(K_inv).dot(Y_train)

    cov_s = K - cov_train.T.dot(K_inv).dot(cov_train)

    return mu_s, cov_s
```

Define the posterior prediction. It takes the observe data, some support data and the kernel parameters in input. It returns after a few computations a mean vector and a covariance matrix.

```python
def log_likelihood(param):
    K = kernel(X_train, X_train,param) + 1e-5*np.eye(len(X_train))
    L = np.linalg.cholesky(K)
    return np.sum(np.log(np.diagonal(L))) + \
            0.5 * Y_train.T.dot(np.linalg.lstsq(L.T, np.linalg.lstsq(L, Y_train)[0])[0]) + \
            0.5 * len(X_train) * np.log(2*np.pi)
```

Compute the negative log-likelihood

```python
def plot_GP(mu, cov, X, X_train=None, Y_train=None, samples=[]):
    X = X.ravel()   # unfold data
    mu = mu.ravel() # unfold data
    uncertainty = 1.96 * np.sqrt(np.diag(cov))  # Compute the uncertainty based on variance

    plt.fill_between(X, mu + uncertainty, mu - uncertainty, alpha=0.1)
    plt.plot(X, mu, label='Mean')
    for i, sample in enumerate(samples):
        plt.plot(X, sample, lw=1, ls='--', label=f'Sample {i+1}')
    if X_train is not None:
        plt.plot(X_train, Y_train, 'rx')
    plt.legend()
```

A function to plot our result. It takes a mean vector mu, a covariance matrix and data to plot in input. It computes the uncertainty as well. You will see exemple of output in the result section of this report.

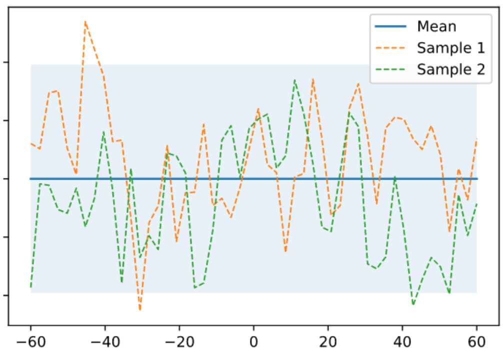
```
X_train = data[:,0].reshape(-1,1)
Y_train = data[:,1]

# STEP 1 : Build prior.
# Our training data are between -50 and 50, so we'll take -60 and 60 to build our prior
X = np.linspace(-60,60,50).reshape(-1,1)
# Build the mean vector and covariance matrix of our prior points
mu = np.zeros(X.shape)
cov = kernel(X,X)   # Covariance is based on the kernel

# Compute and plot samples
samples = np.random.multivariate_normal(mu.ravel(),cov,2)
plot_GP(mu,cov,X,samples=samples)
```

This part of the code build X_train and Y_train from our array data, and then build the prior. I built an array of 50 points between -60 and 60, with a null mean vector, and a covariance matrix derivated from the kernel applied on my vector X. The samples are built as multivariate normal distribution with the defined covariance matrix and mean vector.

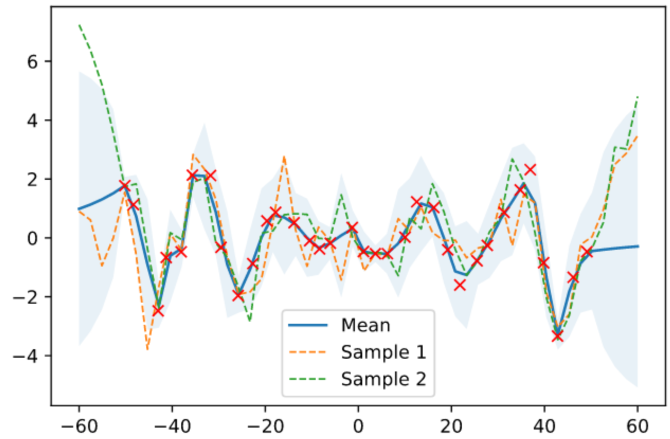
```
77  # param = [l,sigma,alpha] are the parameters of the kernel
78  param=[3,3,3]
79  mu_posterior,cov_posterior = posterior_predictive(X,X_train,Y_train,param)
80  samples_post = np.random.multivariate_normal(mu_posterior.ravel(),cov_posterior,2)
81  plt.figure()
82  plot_GP(mu_posterior,cov_posterior,X,samples=samples_post)
83
84  rez = minimize(log_likelihood,param,bounds=((1e-5,None),(1e-5,None),(1e-5,None)))
85  param = rez['x']
86
87  mu_posterior,cov_posterior = posterior_predictive(X,X_train,Y_train,param)
88  samples_post = np.random.multivariate_normal(mu_posterior.ravel(),cov_posterior,2)
89  plt.figure()
90  plot_GP(mu_posterior,cov_posterior,X,samples=samples_post)
```

From line 78 to line 82 : I basically intialized the parameters arbitrary, and derivated the posterior parameter of the normal distribution. Then I computed two samples from this multivariate distribution and plotted it.
Line 84-85 : I adjusted the parameter of the kernel to minimize the log-likelihood, using scipy.optimise.minimize.
Line 87 to 90 : I did the same as from line 78 to line 82, with the updated parameters for the kernel. The results are better !

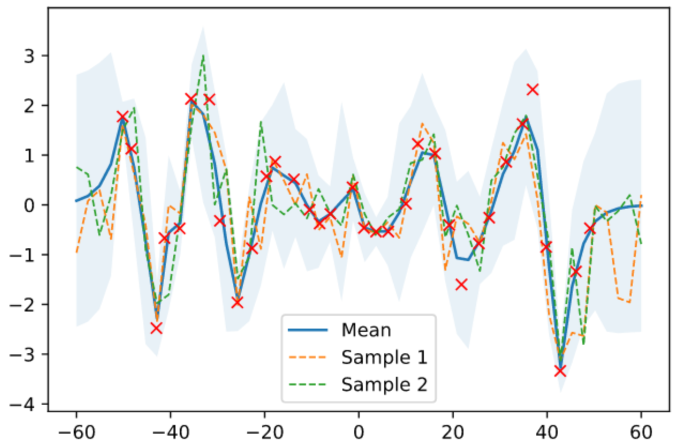
b) Results



Plots of the rational quadratic kernel samples, before observing datas. Taking mean = 0 is ok because it will adjust in the posterior prediction.



Two samples of posterior distribution, with our observed data. The grey region corresponds to the uncertainty. The blue plain line is the mean vector.



Two samples of the posterior distribution after optimizing the parameter with scipy. We observe that the samples fit better the data.

Results of the parameters optimization :

```
[1.26580488e+00, 1.29317786e+00, 1.45866309e+03]
```

II – SVM on MNIST dataset

a) Code

```
import numpy as np
import sys
sys.path.insert(0, 'libsvm-3.24/python')
sys.path.insert(0, 'libsvm-3.24/tools')
from svmutil import *
import grid
from scipy.spatial.distance import *

```

Import usefull libraries, and necessary path to use libsvm. Note : I tried to use the grid.py file to find optimized parameter for kernels, but couldn't manage to make it work, neither to find an accurate documentation on how to use it in python...

```python
Y_train = open('Y_train.csv','r').read().splitlines()
for i in range(len(Y_train)):
    Y_train[i] = float(Y_train[i])
Y_train=np.asarray(Y_train)

X_train = open('X_train.csv','r').read().splitlines()
for i in range(len(X_train)):
    X_train[i] = X_train[i].split(',')
    for j in range(len(X_train[i])):
        X_train[i][j] = float(X_train[i][j])
X_train = np.asarray(X_train)


Y_test = open('Y_test.csv','r').read().splitlines()
for i in range(len(Y_test)):
    Y_test[i] = float(Y_test[i])
Y_test=np.asarray(Y_test)

X_test = open('X_test.csv','r').read().splitlines()
for i in range(len(X_test)):
    X_test[i] = X_test[i].split(',')
    for j in range(len(X_test[i])):
        X_test[i][j] = float(X_test[i][j])
X_test = np.asarray(X_test)
```

Import the data from the csv files, and preprocess it.

```python
cs = [1e-4,1e-3,1e-2,1e-1,1e0]
for c in cs:
    prob = svm_problem(Y_train,X_train)
    param = svm_parameter('-t 0 -c '+str(c))
    m_linear = svm_train(prob,param)
    print("Linear Model result with c = {}:".format(c))
    p_label, p_acc, p_val = svm_predict(Y_test,X_test, m_linear)
    # p_acc contains : accuracy, mean squared error and squared correlation coefficient
deg = [2,3,4,5]
cs = [10,100,1000,10000]
for d in deg:
    for c in cs:
        param = svm_parameter('-t 1 -c '+str(c)+' -d '+str(d))
        m_poly = svm_train(prob,param)
        print("Polynomial Model result with c = {} and d = {} :".format(c,d))
        p_label, p_acc, p_val = svm_predict(Y_test,X_test, m_poly)

gammas = [1e-3,1e-2,0.1]
cs = [1e1,1e2,1e3,1e4]
for g in gammas:
    for c in cs:
        param = svm_parameter('-t 2 -c '+str(c)+' -g '+str(g))
        m_rbf = svm_train(prob,param)
        print("RBF Model result for c = {} and g = {} :".format(c,g))
        p_label, p_acc, p_val = svm_predict(Y_test,X_test, m_rbf)
```

For each kernel type (Linear, Polynomial and RBF), create the problem with svm_problem, adjust parameter with svm_parameter and train the model with svm_train. Then I test the model on the test set with svm_predict. I do this for several values of the parameter, in order to find the most accurate parameter.

```
def linear_kernel(x,y,c):
    res = x@y.T + c
    return res

def rbf_kernel(x,y,gamma):
    temp = cdist(x,y,'euclidean')
    return np.exp(-gamma*temp)


cs=[1e-4,1e-2,1,1e2]
gammas = [1e-3,1e-2,0.1]

for c in cs:
    for gamma in gammas:
        K = linear_kernel(X_train,X_train,c)
        K += rbf_kcernel(X_train,X_train,gamma)
        K/=2
        KK = linear_kerne X_test: list t,c)
        KK += rbf_kernel(X_test,X_test,gamma)
        K/=2
        model = svm_train(Y_train,K,'-t 4')
        e,f,g = svm_predict(Y_test,KK,model)
```

I defined two kernels that we want to combine (LINEAR and RBF), and took the average of the result of it on both X_train and X_test. Then I used svm_train to train the model using this kernel matrix, and svm_predict to test the model. I made two loop in order to try it on different parameter, in order to find the best one for this kernel.

b) Results

```
Linear Model result with c = 0.0001:
Accuracy = 89.92% (2248/2500) (classification)
Linear Model result with c = 0.001:
Accuracy = 94.68% (2367/2500) (classification)
Linear Model result with c = 0.01:
Accuracy = 95.96% (2399/2500) (classification)
Linear Model result with c = 0.1:
Accuracy = 95.8% (2395/2500) (classification)
Linear Model result with c = 1.0:
Accuracy = 95.08% (2377/2500) (classification)
```

These are the results with the linear model kernel. We have pretty good results, and the best parameter for it are with c = 0,01. The execution time is really fast considering the size of our dataset !

```
Polynomial Model result with c = 10 and d = 2 :
Accuracy = 95.2% (2380/2500) (classification)
Polynomial Model result with c = 100 and d = 2 :
Accuracy = 97.68% (2442/2500) (classification)
Polynomial Model result with c = 1000 and d = 2 :
Accuracy = 97.72% (2443/2500) (classification)
Polynomial Model result with c = 10000 and d = 2 :
Accuracy = 97.68% (2442/2500) (classification)
Polynomial Model result with c = 10 and d = 3 :
Accuracy = 79.72% (1993/2500) (classification)
Polynomial Model result with c = 100 and d = 3 :
Accuracy = 93.48% (2337/2500) (classification)
Polynomial Model result with c = 1000 and d = 3 :
Accuracy = 97.6% (2440/2500) (classification)
Polynomial Model result with c = 10000 and d = 3 :
Accuracy = 97.4% (2435/2500) (classification)
Polynomial Model result with c = 10 and d = 4 :
Accuracy = 31.8% (795/2500) (classification)
Polynomial Model result with c = 100 and d = 4 :
Accuracy = 70.76% (1769/2500) (classification)
Polynomial Model result with c = 1000 and d = 4 :
Accuracy = 90.32% (2258/2500) (classification)
Polynomial Model result with c = 10000 and d = 4 :
Accuracy = 96.08% (2402/2500) (classification)
Polynomial Model result with c = 10 and d = 5 :
Accuracy = 22.56% (564/2500) (classification)
Polynomial Model result with c = 100 and d = 5 :
Accuracy = 32.56% (814/2500) (classification)
Polynomial Model result with c = 1000 and d = 5 :
Accuracy = 63.92% (1598/2500) (classification)
Polynomial Model result with c = 10000 and d = 5 :
Accuracy = 85.08% (2127/2500) (classification)
```

Here are the results of the polynomial model. I took a wide range of value for c and d, but had the best results for Degree = 2 and C = 1000. The high value of C avoids overfitting. I also had good results with degree = 3 and c = 1000. I didn't change the value of gamma, so it is by default 1/num_features. I also left empty the coef0 parameter. I would have changed it if my results were bad, but they are already really good by letting this 2 parameters by default.

```
RBF Model result for c = 10.0 and g = 0.001 :
Accuracy = 96.2% (2405/2500) (classification)
RBF Model result for c = 100.0 and g = 0.001 :
Accuracy = 96.84% (2421/2500) (classification)
RBF Model result for c = 1000.0 and g = 0.001 :
Accuracy = 96.56% (2414/2500) (classification)
RBF Model result for c = 10000.0 and g = 0.001 :
Accuracy = 96.56% (2414/2500) (classification)
RBF Model result for c = 10.0 and g = 0.01 :
Accuracy = 98.2% (2455/2500) (classification)
RBF Model result for c = 100.0 and g = 0.01 :
Accuracy = 98.16% (2454/2500) (classification)
RBF Model result for c = 1000.0 and g = 0.01 :
Accuracy = 98.16% (2454/2500) (classification)
RBF Model result for c = 10000.0 and g = 0.01 :
Accuracy = 98.16% (2454/2500) (classification)
RBF Model result for c = 10.0 and g = 0.1 :
Accuracy = 91% (2275/2500) (classification)
RBF Model result for c = 100.0 and g = 0.1 :
Accuracy = 91% (2275/2500) (classification)
```

RBF gave me the best results overall, with over 98 % accuracy ! I obtained these results with the following parameters : c = 10 and g = 0,01.

Then I tried to mix 2 kernels but had the following really bad results..

```
Accuracy = 22.36% (559/2500) (classification)
Accuracy = 22.36% (559/2500) (classification)
Accuracy = 21.64% (541/2500) (classification)
Accuracy = 22.36% (559/2500) (classification)
Accuracy = 22.36% (559/2500) (classification)
Accuracy = 21.72% (543/2500) (classification)
Accuracy = 22.64% (566/2500) (classification)
Accuracy = 22.68% (567/2500) (classification)
Accuracy = 22.36% (559/2500) (classification)
Accuracy = 20.04% (501/2500) (classification)
Accuracy = 20.04% (501/2500) (classification)
Accuracy = 20.04% (501/2500) (classification)
```

I tried many other parameters but couldn’t find some that gave a decent accuracy. Seeing the kernel matrix, I felt like the linear kernel had too much weight compared to the rbf kernel. But even tho I tried to decrease its weight in order to have same kind of value in both kernels matrix, I still had similar results. I don’t really have any explanation to explain these results… I have same results as if I was picking randomly the label of every image, which mean the model didn’t train at all on the data. Maybe I have a wrong use of LIBSVM with user kernel, but I tried to do as some examples I found on internet.