ML HW05

Using python 3.5.3

■ DATA PREPROCESS

作業提供的 mnist 檔案已經將資料 scale 至 [0,1]，因此不用再對此做處理。以下只將資料轉換成 LIBSVM 所需要的格式

<label> <index>:<value> <index>:<value>…

```
import pandas as pd
#LIBSVM format convert
#<label> <index>:<value> <index:<value>...
if __name__=="__main__":
    #note that LIBSVM needs the <index> to start from 1
    #read pixel csv file, with header that starts with 1
    train_x = pd.read_csv("../../X_train.csv",header=0,names=range(1,785))
    #add indexs <index>:<value>
    for col in train_x:
        train_x[col]=col.astype(str)+':'+train_x[col].astype(str)
    #read label(class) file
    train_y = pd.read_csv("../../T_train.csv", header=0, names=[0])
    #concat to form <label> <index>:<value>....
    train_data=pd.concat([train_y, train_x], axis=1)
    #save without extra pandas dataframe labels
    train_data.to_csv('train_mnist',sep=' ',index=False,header=False)
    print('Saved train_mnist')

    #same thing performed on test_data
    test_x = pd.read_csv("../../X_test.csv",header=0,names=range(1,785))
    for col in test_x:
        test_x[col]=col.astype(str)+':'+test_x[col].astype(str)
    test_y = pd.read_csv("../../T_test.csv", header=0, names=[0])
    test_data=pd.concat([test_y, test_x], axis=1)
    test_data.to_csv('test_mnist',sep=' ',index=False,header=False)
    print('Saved test_mnist')
```

- A QUICK IMPLEMENT

  svm_read_problem(file_name)

  會將檔案讀入後切割成它需要的格式(好像繞了一圈)

  svm_train(labels,data,flags)

  會根據 flags 訓練 SVM，-c 是 cost，-t 是對應的 kernel。

  在此選擇的值沒有任何的意義，只是快速的測試。

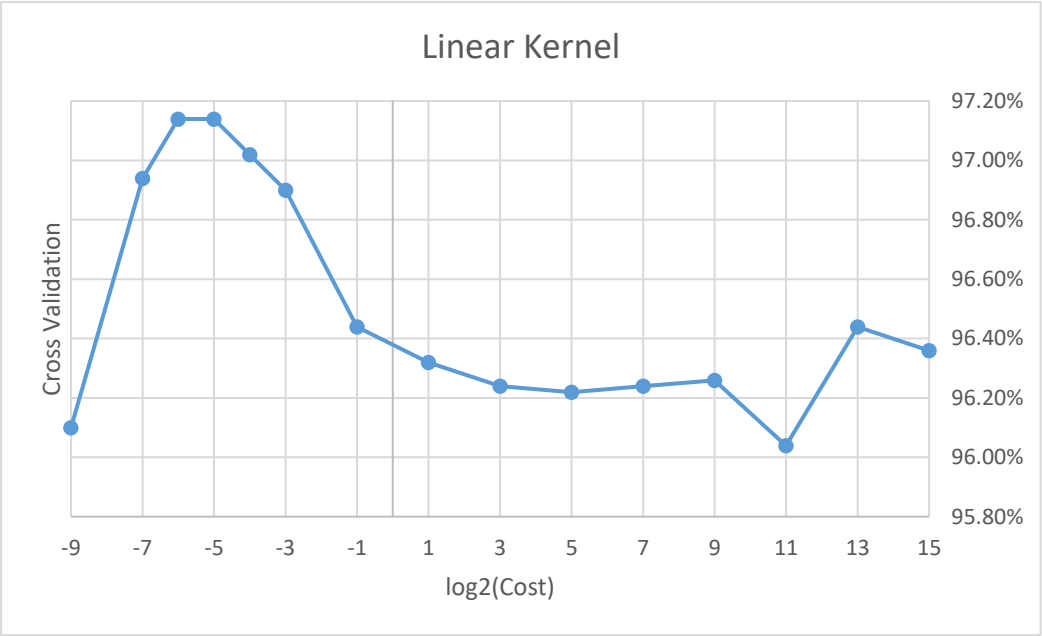
```
from svmutil import *
kernel_dict={0:'Linear',1:'Polynomial',2:'RBF',3:'Sigmoid'}
train_y, train_x = svm_read_problem('train_mnist')
test_y, test_x = svm_read_problem('test_mnist')
#a quick implement
m=list()
for i in range(3):
    m.append(svm_train(train_y, train_x, '-c 4 -t '+str(i)))
for i in range(3):
    print("Prediction with kernel {}".format(kernel_dict[i]))
    p_label, p_acc, p_val = svm_predict(test_y,test_x, m[i])
```

```
Prediction with kernel Linear
Accuracy = 94.998% (2374/2499) (classification)
Prediction with kernel Polynomial
Accuracy = 65.1861% (1629/2499) (classification)
Prediction with kernel RBF

Accuracy = 96.1585% (2403/2499) (classification)
```

可以看到 Polynomial 在這些數值(default degree=3)下，表現比較差。Linear 表現得出乎我意料的話，我原本以為這是沒辦法用 linear kernel 去做 classify 的問題，應該是因為圖片只有 0~4，並沒有常常混淆的 089 之類的。但是在此的數據沒有什麼意義，若 linear 表現的不錯，polynomial 在調整後應該也可以得到不錯的結果。

■ LINEAR KERNEL



(ten fold cross validation)

Cost 越小，代表著對 training data 更 soft，有可能出現 underfitting 的狀況。Cost 越大就代表著越嚴格，有可能出現 overfitting。可以觀察到無論對 Cost 做多少調整其 cross validation 值都在 96%以上，代表著他有一個明顯的 linear boundary。在此可以預測後面的 kernel 都可以得到不錯的結果，若 polynomial degree 越大，則會需要越大的 C 來維持這個 boundary。

我們取 cross validation 最好的 Cost=0.015，得到

■ 95.8784% (2396/2499) (classification)

仍然有相當好的表現。

```
#searching through linear
#[-5,-3,-1.....,15]
c_power=range(-5,17,2)
#the pow of 2
c_grid=[2**p for p in c_power]
#c_grid=[0.015,0.02,0.025,0.03]
for c_value in c_grid:
    print("C value: {}".format(c_value))
    svm_train(train_y, train_x, '-t 0 -v 10 -q -c '+str(c_value))
```

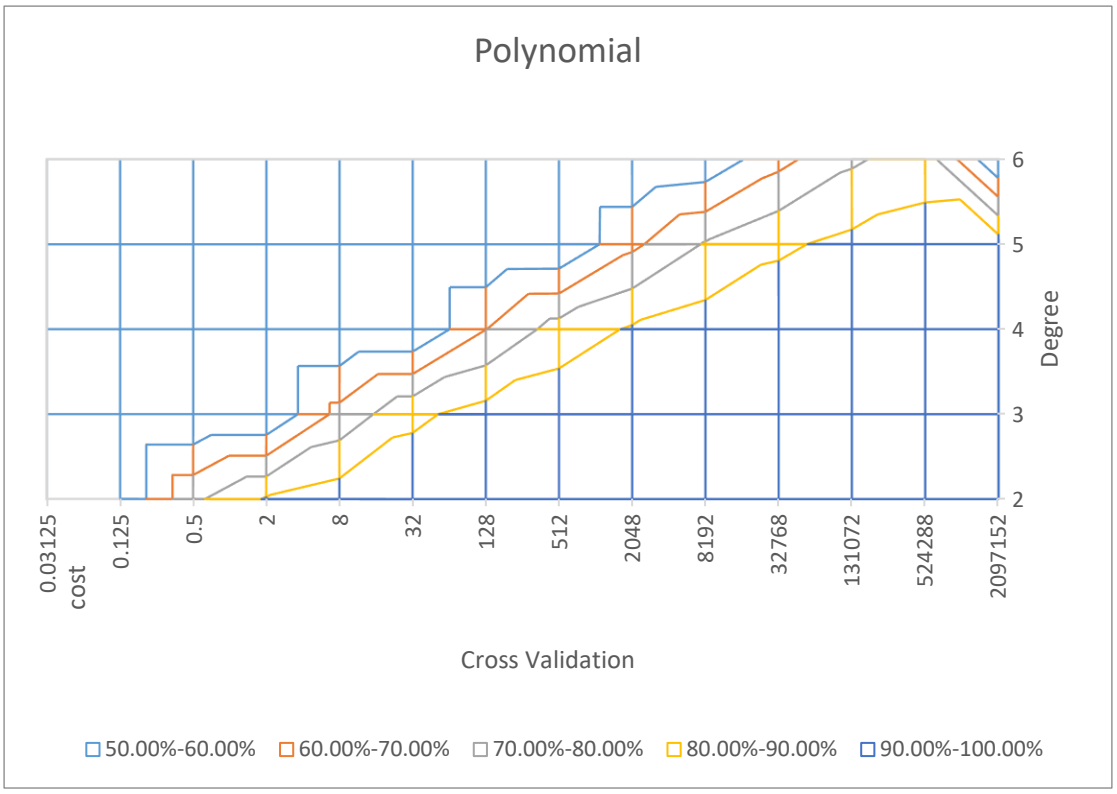
■ ‘-t 0 –v 10 –q –c ‘+str(c_value)

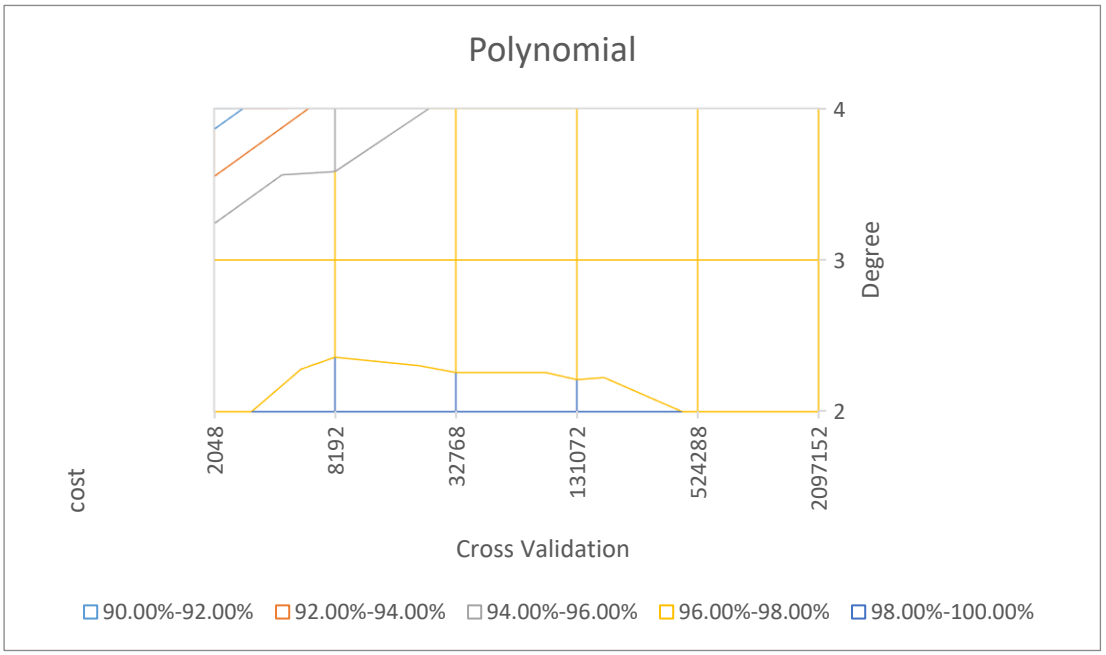
用 linear kernel(0)，進行 10 fold cross validation 不要輸出過程，cost 值 c_value

svm_predict(test_y,test_x,m)，拿 m 去預測 test data。

```
m=svm_train(train_y, train_x, '-t 0 -q -c 0.015')
#give prediction
p_label, p_acc, p_val = svm_predict(test_y,test_x,m)
```

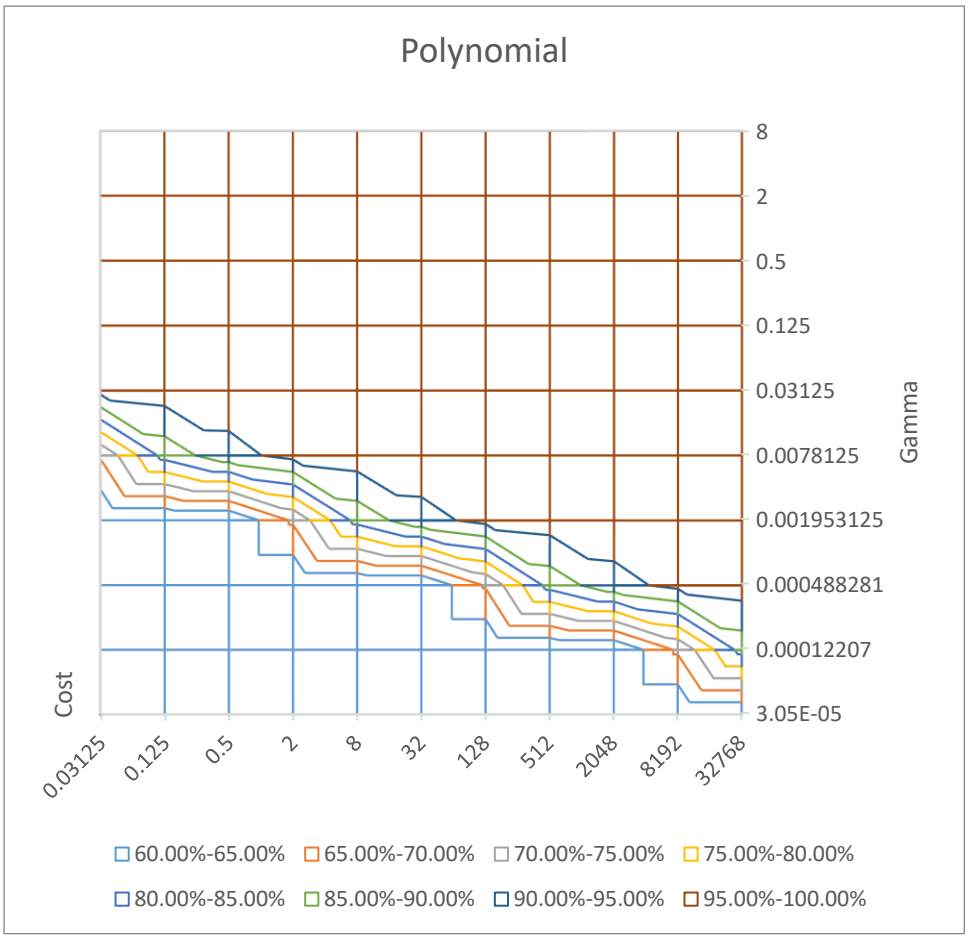
■ POLYNOMIAL KERNEL





5-fold gamma=1/784

可以看到越往右下角走，degree 變小，C 變大 Cross validation 才會變好，下圖是右下角的放大部分。這在上一部份有提及，因為有很明顯的 linear boundary，若用大的 degree 會容易 underfit，但若用較大的 C 限制住 boundary 的話就可以維持。



5-fold degree=3

可以觀察到在 gamma 越大的時候，cross validation 才會比較好。Gamma 的增加會使 kernel 值直接增加，加強 boundary

基本上和前面一樣，c_grid 自由調整，-t 是 poly kernel -d 是 degree 值。

```
34  c_power=range(23,29,2)
35  c_grid=[2**p for p in c_power]
36  degree=range(5,7)
37  for d_value in degree:
38      for c_value in c_grid:
39          print("C: {} Degree: {}".format(c_value,d_value))
40          svm_train(train_y, train_x, '-t 1 -v 5 -q -c '+str(c_value)+' -d '+str(d_value))
```

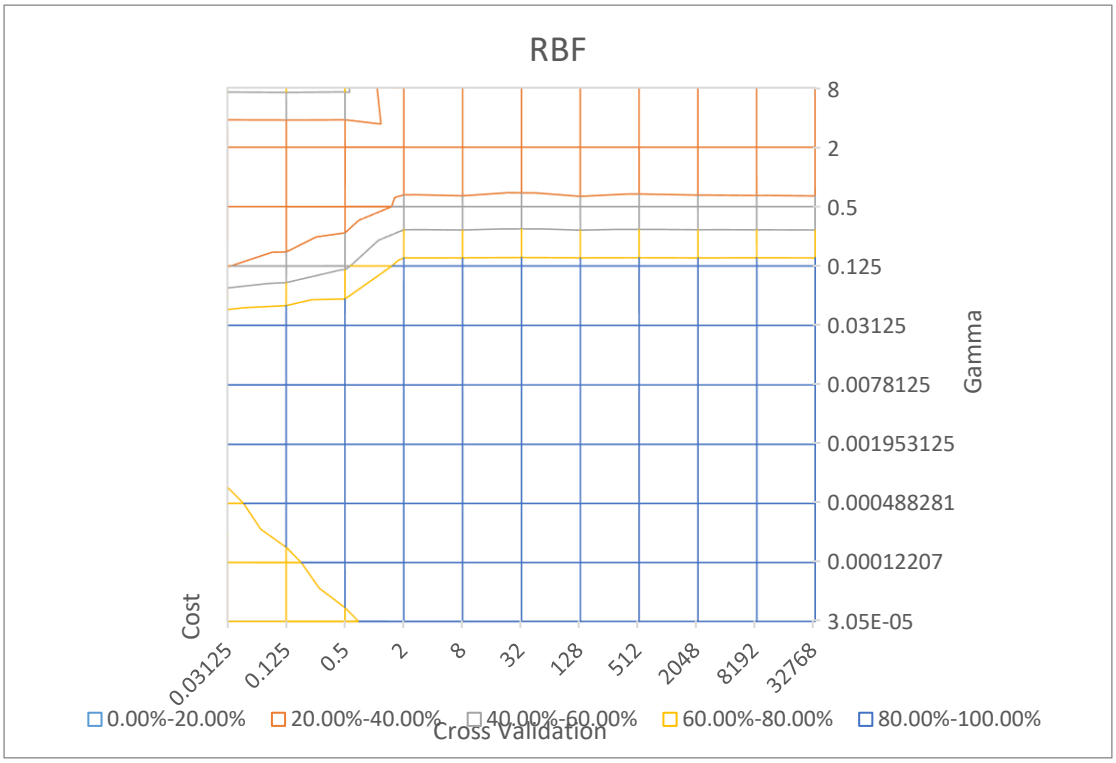
我們選用 c=8192,degree=2

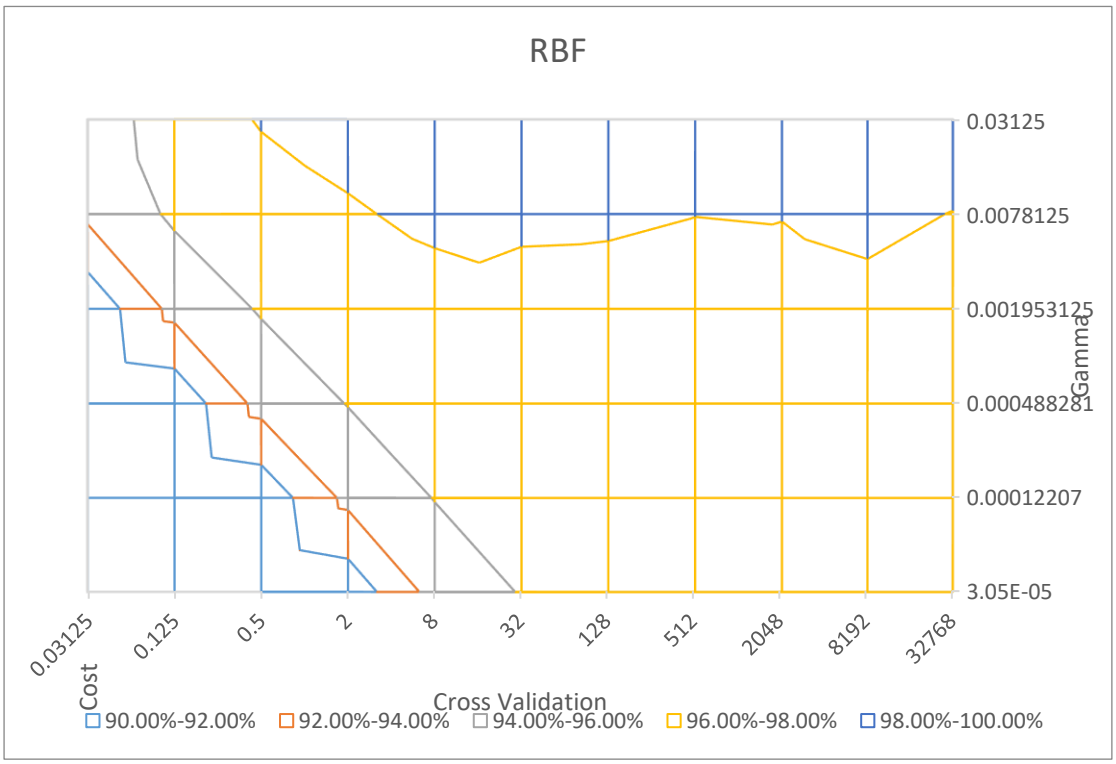
Accuracy = 97.6791% (2441/2499) (classification)

(ACC,MSE,SCC)

97.67907162865146 0.06482593037214886 0.9677605947418593

■ RBF KERNEL





這是 RBF 的圖，記得這是個 linear kernel 就可以解決的問題，把這樣的問題投影到更多維的空間時雖然”理論上”可以至少達到 linear kernel 的效果，但這是建立

在 Cost 跟 Gamma 調到好的時候。可以觀察到在這種狀況下他的圖已經不像前面那麼簡單偏向一角，而是起起伏伏。可以預見最後一個應該會更難找到。
一樣，只是將 –t 改 2。

```
43  c_power=range(-1,17,4)
44  gamma_power=range(-7,-1,2) #to 2**-3
45  c_grid=[2**p for p in c_power]
46  gamma_grid=[2**p for p in gamma_power]
47  for gamma_value in gamma_grid:
48      for c_value in c_grid:
49          print("C: {} Gamma: {}".format(c_value,gamma_value))
50          svm_train(train_y, train_x, '-t 2 -v 5 -q -c '+str(c_value)+' -g '+str(gamma_value))
```

我們使用最佳值 C=8,Gamma=0.03125
Accuracy = 98.5194% (2462/2499) (classification)
(ACC,MSE,SCC)
98.51940776310524 0.05002000800320128 0.9750680747163384

- COMPARE

目前表現上 RBF 98.5194% > POLYNOMIAL 97.6791% > LINEAR 95.8784%
在適當的調整 parameter 後，較為複雜的 feature space 可以找到更好的分類法。

■ LINEAR+RBF

結果非常的悽慘，以下直接列表格。

C\Gamma	7.63E-06	3.05E-05	0.000122	0.000488	0.001953	0.007813	0.03125	0.125
0.000488	20.00%	20.00%	20.00%	20.00%	20.00%	20.00%	20.00%	20.00%
0.007813	20.00%	20.00%	20.00%	20.00%	20.00%	20.00%	20.00%	20.00%
0.125	20.00%	20.00%	20.00%	20.00%	22.86%	23.98%	31.23%	22.54%
0.5	20.00%	20.00%	20.00%	20.00%	20.00%	21.24%	29.07%	20.00%
8	20.00%	20.00%	20.00%	20.00%	20.00%	25.65%	20.16%	21.38%
128	20.00%	20.00%	20.00%	20.00%	20.00%	23.86%	27.15%	21.56%
2048	20.00%	20.00%	20.00%	20.00%	20.00%	23.72%	27.45%	20.26%
32768	20.00%	20.00%	20.00%	20.00%	20.00%	28.09%	20.38%	

C\Gamma	0.01	0.02	0.03
0.5	22.94%	20.16%	23.28%
8	26.79%	29.27%	22.46%
128	25.49%	27.85%	25.07%
2048	23.46%	20.12%	25.39%

32768	27.73%	26.09%	29.05%
524288	30.07%	30.59%	23.3647
2097152	24.38%	25.71%	
8388608	26.79%	29.67%	
33554432	25.49%	20.16%	
134217728	23.46%	29.27%	

距離更遠的 grid 都有搜尋了，以上只是列出比較有起伏的位置。

要使用自定義的 kernel 的話，需要更改 svm.cpp。我將 sigmoid kernel 的部分直接挪用來使用，要調整的部分主要是 kernel_function 和 k_function。

Kernel_function

```
221     const svm_node **x;
222     double *x_square;
223
224     // svm_parameter
225     const int kernel_type;
226     const int degree;
227     const double gamma;
228     const double coef0;
229
230     static double dot(const svm_node *px, const svm_node *py);
231     double kernel_linear(int i, int j) const
232     {
233         return dot(x[i],x[j]);
234     }
235     double kernel_poly(int i, int j) const
236     {
237         return powi(gamma*dot(x[i],x[j])+coef0,degree);
238     }
239     double kernel_rbf(int i, int j) const
240     {
241         return exp(-gamma*(x_square[i]+x_square[j]-2*dot(x[i],x[j])));
242     }
243     double kernel_sigmoid(int i, int j) const
244     {
245         return dot(x[i],x[j])+exp(-gamma*(x_square[i]+x_square[j]-2*dot(x[i],x[j])));
246         //return tanh(gamma*dot(x[i],x[j])+coef0);
247     }
248     double kernel_precomputed(int i, int j) const
249     {
250         return x[i][(int)(x[j][0].value)].value;
251     }
```

K_function

```
double Kernel::k_function(const svm_node *x, const svm_node *y,
              const svm_parameter& param)
{
    switch(param.kernel_type)
    {
        case LINEAR:
            return dot(x,y);
        case POLY:
            return powi(param.gamma*dot(x,y)+param.coef0,param.degree);
        case RBF:
        {
        case SIGMOID:
        {
        case PRECOMPUTED:  //x: test (validation), y: SV
            return x[(int)(y->value)].value;
        default:
            return 0;  // Unreachable
    }
}
```

```
        case SIGMOID:
        {
            double sum = 0;
            while(x->index != -1 && y->index !=-1){
                if(x->index == y->index){
                    double d = x->value - y->value;
                    sum += d*d;
                    ++x;
                    ++y;
                }
                else{
                    if(x->index > y->index){
                        sum += y->value * y->value;
                        ++y;
                    }
                    else{
                        sum += x->value * x->value;
                        ++x;
                    }
                }
            }
            while(x->index != -1){
                sum += x->value * x->value;
                ++x;
            }
            while(y->index != -1){
                sum += y->value * y->value;
                ++y;
            }
            return exp(-param.gamma*sum)+dot(x,y);
            //return tanh(param.gamma*dot(x,y)+param.coef0);
        }
```

另外還有一個要注意的是 x_square(他算平方的函式)，要讓”sigmoid”也能使用。

```
    //if(kernel_type == RBF)
    if(kernel_type == RBF || kernel_type == SIGMOID)
    {
        x_square = new double[l];
        for(int i=0;i<l;i++)
            x_square[i] = dot(x[i],x[i]);
    }
    else
        x_square = 0;
```

接下來就用 nmake 產 shared library libsvm.dll，替換掉原來在 windows 資料夾內的。

程式碼和前面相似，只是注意-t 3 原先指向 sigmoid，現在是指向 linear+RBF

```
#through LINEAr + RBF
c_power=range(21,29,2)
gamma_power=range(-17,-1,2)
c_grid=[2**p for p in c_power]
gamma_grid=[0.01,0.02]
#gamma_grid=[2**p for p in gamma_power]
for gamma_value in gamma_grid:
    for c_value in c_grid:
        print("C: {} Gamma: {}".format(c_value,gamma_value))
        svm_train(train_y, train_x, '-t 3 -v 5 -q -c '+str(c_value)+' -g '+str(gamma_value))
```

用 C=524288,Gamma=3.05E-05

Accuracy = 20.008% (500/2499) (classification)

20.00800320128051 5.995998399359744 nan

慘不忍睹，但是值得注意的是 500/2499，這代表著他只產生單一一個 label。這也可以解釋前面的大量的 20%是從哪裡來的。