0780804 – 高黃江
Hoang-Giang, Cao

HOME WORK 5
MACHINE LEARNING
Pratice Support Vector Machine

1) Use difference kernel functions (by using build-in kernels) (no cross validation)

a) Linear

setting default built-in Linear Kernel (c = 1) & run in test set

```
param = svm_parameter("-t 0 -q")
m = svm_train(prob, param)
p_label, p_acc, p_val = svm_predict(TestLabel, TestData, m)
```

Accuracy = **95.08%** (2377/2500) (classification)

b) Polynomial

setting default built-in Polynomial Kernel (c = 1 , d = 3, gamma = 0, r = 0)

```
param = svm_parameter("-t 1 -q")
m = svm_train(prob, param)
p_label, p_acc, p_val = svm_predict(TestLabel, TestData, m)
```

Accuracy = **34.68%** (867/2500) (classification)

c) RBF

setting default built-in RBF Kernel (gamma = 0)

```
param = svm_parameter("-t 2 -q")
m = svm_train(prob, param)
p_label, p_acc, p_val = svm_predict(TestLabel, TestData, m)
```

Accuracy = **95.32%** (2382/2500) (classification)

d) Comparison default kernels

Linear	Polynomial	RBF
95.08%	34.68%	95.32%

2) C-SVC & grid-search, cross-validation

a) Linear, cross-validation with 3-folds, searching for variable c

<code>

```
gridSearchLinear = True
if (gridSearchLinear == True) :
        bestc = -1
        bestCrossGrid = -1
        logSeting= []
        print ("RUNNING gridSearchLinear")
        for log2c in range (-8,1,1):
                print (2**log2c)
                param = svm_parameter("-q -t 0 -v 3 -c " + str(2**log2c) )
                m = svm_train(prob, param)
                logSeting.append([2**log2c,m])
                if (m > bestCrossGrid) :
                        bestCrossGrid = m
                        bestc = 2**log2c

        for log in logSeting:
                print (log)

        # run on the best setting
        param = svm_parameter("-q -t 0 -c " + str(bestc) )
        m = svm_train(prob, param)
        p_label, p_acc, p_val = svm_predict(TestLabel, TestData, m)
```

Result of findding parameter c, performance on cross-validation

0.00390625	0.0078125	0.015625	0.03125	0.0625	0.125	0.25	0.5	1
96.36	96.52	**96.90**	96.76	96.86	96.84	96.62	96.18	96.22

- **c = 0.015625** give the best performance on cross-validation
- Using the parameter that have the best performance in cross-validation on Test set

- Accuracy = **95.92%** (2398/2500) (classification)
- The accuracy is slightly higher than default setting
- The range of c is based on the suggestion of libsvm guide. We firstly do the spare search to find a range of c, then make a fine-turn to get the better result.

b) Polynomial, cross-validation with 3-folds, searching for variable c, gamma, d

<code>

```
gridSearchPolinomial =  True
if (gridSearchPolinomial == True) :
   bestc = -1
   bestd = -1
   bestg = -1
   bestCrossGrid = -1
   logSeting= []
   print ("RUNNING gridSearchPolinomial")

   for d in range (1,5,1) :
     for log2g in range (-2,2,1):
       for log2c in range (-8,1,1):
         print (2**log2c,2**log2g , d)
         param = svm_parameter("-q -t 1 -v 3 -c " + \
                               str(2**log2c) +" -g " + str (2**log2g) + " -d " + str (d) )
         m = svm_train(prob, param)
         logSeting.append([2**log2c,2**log2g,d,m])
         if (m > bestCrossGrid) :
           bestCrossGrid = m
           bestc = 2**log2c
           bestd = d
           bestg = 2**log2g

   for log in logSeting:
     print (log)
   print ("param: -q -t 1 -c " + str(bestc) +" -g " + str (bestg) + " -d " + str (bestd))
  # run on the best setting
   param = svm_parameter("-q -t 1 -c " + str(bestc) +" -g " + str (bestg) + " -d " + str
(bestd) )
   m = svm_train(prob, param)
   p_label, p_acc, p_val = svm_predict(TestLabel, TestData, m)
```

Result of finding parameter c and gamma, most of the high result show when d = 2, performance on cross-validation

gamma C	0.25	0.5	1	2
0.00390625	97.68	97.8	97.88	97.78
0.0078125	98.08	98.08	97.96	97.82
0.015625	97.84	97.9	97.92	97.94
0.03125	98	97.96	97.96	98
0.0625	98	97.82	97.82	97.88
0.125	97.86	97.98	97.96	97.9
0.25	97.9	97.7	97.86	97.78
0.5	98.1	97.98	98	97.8
1	97.74	97.96	97.8	98.06

grid-search on (C,gamma) when d = 2

(There are still many table result with difference d, but the result is not better than d =2)

- c = 0.5 ; g = 0.25 ; d = 2 give the best performance on cross-validation with the accuracy is 98.1%
- Using the parameter that have the best performance in cross-validation on Test set
- Accuracy = **97.68%** (2442/2500) (classification)
- The accuracy is slightly higher than default setting
- The polynomial kernel require more time to train, compare to the linear kernel

c) RBF ,cross-validation with 3-folds, searching for variable c, gamma

<code>

```
gridSearchRBF== True
if (gridSearchRBF == True) :
       bestc = -1
       bestd = -1
       bestg = -1
```

```
        bestCrossGrid = -1
        logSeting= []
        print ("RUNNING gridSearchRBF")
        count  = 0
        print ( (7*12) )
        for log2g in range (-5,2,1):
                for log2c in range (-3,9,1):
                        count+=1
                        print (count,2**log2c,2**log2g )
                        param = svm_parameter("-q -t 2 -v 3 -c " + str(2**log2c) +" -g " + str
(2**log2g) )
                        m = svm_train(prob, param)
                        logSeting.append([2**log2c,2**log2g,m])
                        if (m > bestCrossGrid) :
                                bestCrossGrid = m
                                bestc = 2**log2c
                                bestg = 2**log2g


        for log in logSeting:
                print (log)
        print ("param: -q -t 2 -c " + str(bestc) +" -g " + str (bestg) )
        # run on the best setting
        param = svm_parameter("-q -t 2 -c " + str(bestc) +" -g " + str (bestg) )
        m = svm_train(prob, param)
        p_label, p_acc, p_val = svm_predict(TestLabel, TestData, m)
```

Result of finding parameter c and gamma, performance on cross-validation set

gamma C	0.03125	0.0625	0.125	0.25	0.5	1	2
0.125	96.94	84.48	47.34	28.32	21.42	20.56	20.34
0.25	97.64	92.7	49.08	33.9	21.62	20.72	20.38
0.5	97.96	96.94	54.66	39.96	25.18	20.58	20.26
1	98.26	97.78	83.82	62.48	45.68	30.56	23.86
2	98.4	97.76	84.86	65.48	44.58	31.02	24.92
4	98.52	97.9	85.02	65.52	45.14	31.82	25.38
8	98.46	97.68	85.22	65.42	44.9	31.78	26.08
16	98.36	97.84	85.38	65.64	45.48	31.68	25.62
32	**98.64**	97.86	84.82	64.46	45.2	31.58	24.82
64	98.44	97.74	85.02	65.36	44.72	31.02	25.6
128	98.48	97.82	84.66	65.64	44.04	32.62	25.06
256	98.56	97.76	84.96	64.86	44.6	31.04	25.04

- c = 32 ; g = 0.03125 give the best performance on cross-validation with the accuracy is 98.64%

- Using the parameter that have the best performance in cross-validation on Test set
- Accuracy = **98.52%** (2463/2500) (classification)
- Higher than default setting. Increasing the gamma then will give worst result
- The time to train with RBF kernel is slower than the linear and the polynomial kernels.

Since later, when we precompute kernel, the time for training is reduced

d) Comparision using grid-search

Running on testing set with the setting that give the best performance on cross-validation

Linear	Polynomial	RBF
95.92%	97.68%	98.52%

3) User-defined Linear Kernel + RBF kernel, comparing its performance

a) Linear user-defined

Compute Linear kernel, defined as K(Xi , Xj) = Xi.T Xj

```
<code> Calculate Linear kernel
        nTrain = len(TrainLabel)
nTest = len(TestLabel)
# LINEAR KERNEL
TrainData = np.array(TrainData)
TestData = np.array(TestData)

LinearKernelTrain = TrainData.dot(TrainData.T)
LinearKernelTest = TestData.dot(TrainData.T)

#add index to the first col
index = range (1,nTrain+1)
LinearKernelTrain = np.column_stack([index, LinearKernelTrain])
```

```
LinearKernelTrain = (LinearKernelTrain).tolist()

#add index to the first col
index = range (1,nTest+1)
LinearKernelTest = np.column_stack([index, LinearKernelTest])
LinearKernelTest = (LinearKernelTest).tolist()
```

b) RBF user-defined

Compute RBF kernel, defined as $\exp(-\gamma\|Xi - Xj\|_2), \gamma > 0$

```
<code> Calculate RBF kernel, g = 0.03125, from the previous experiment
        g =  0.03125
# RBF KERNEL
#Train Side
#L2 norm each point
difDist= np.zeros((nTrain,nTrain))

for i in range (nTrain):
        for j in range (i,nTrain,1):
                difDist[i,j] = np.linalg.norm( TrainData[i] -TrainData[j])
                difDist[j,i] = difDist[i,j]

RBFKernelTrain = np.zeros((nTrain,nTrain))
#RBF exp((-gamma)(|| x- y || )
for i in range (nTrain):
        for j in range (i,nTrain,1):
                RBFKernelTrain[i,j] =  exp((-g)*difDist[i,j])
                RBFKernelTrain[j,i] =  RBFKernelTrain[i,j] # symmetric

#Test Side
#L2 norm each point
difDistTest= np.zeros((nTest,nTrain))

for i in range (nTest):
        for j in range (nTrain):
                difDistTest[i,j] = np.linalg.norm(TestData[i]-TrainData[j])

RBFKernelTest = np.zeros((nTest,nTrain))
#RBF exp((-gamma)(|| x- y ||
for i in range (nTest):
        for j in range (nTrain):
                RBFKernelTest[i,j] = exp((-g) *difDistTest[i,j])

#add index to the first col
index = range (1,nTrain+1)
RBFKernelTrain = np.column_stack([index, RBFKernelTrain])
RBFKernelTrain = (RBFKernelTrain).tolist()

#add index to the first col
index = range (1,nTest+1)
RBFKernelTest = np.column_stack([index, RBFKernelTest])
RBFKernelTest = (RBFKernelTest).tolist()
```

c) Combination

Combine 2 valid kernels to be a new kernel.

Linear + RBF kernel = $K(Xi, Xj) = Xi.T \times j + \exp(-\gamma\|Xi - Xj\|_2), \gamma > 0.$

```
<code> Calculate Linear + RBF kernel
        #Linear + RFB
LRBFKernelTrain = np.zeros((nTrain,nTrain))

for i in range (nTrain):
        for j in range (i,nTrain,1):
                LRBFKernelTrain[i,j] = LinearKernelTrain[i][j+1] + RBFKernelTrain[i][j+1]
                LRBFKernelTrain[j,i] =  LRBFKernelTrain[i,j] #symmetric

LRBFKernelTest = np.zeros((nTest,nTrain))

for i in range (nTest):
        for j in range (1,nTrain,1):
                LRBFKernelTest[i,j] = LinearKernelTest[i][j+1] + LinearKernelTest[i][j+1]

#add index to the first col
index = range (1,nTrain+1)
LRBFKernelTrain = np.column_stack([index, LRBFKernelTrain])
LRBFKernelTrain = (LRBFKernelTrain).tolist()

#add index to the first col
index = range (1,nTest+1)
LRBFKernelTest = np.column_stack([index, LRBFKernelTest])
LRBFKernelTest = (LRBFKernelTest).tolist()
```

d) Result

<code> Runing on testing set, c = 32 as previous setting

```
        print("LINEAR + RBF MANUAL")
prob  = svm_problem(TrainLabel, LRBFKernelTrain, isKernel=True)
print ("Param " + "-t 4 -c 32 -q")
param = svm_parameter("-t 4 -c 32 -q")
m = svm_train(prob, param)
p_label, p_acc, p_val = svm_predict(TestLabel, LRBFKernelTest, m)
```

- Accuracy = 95.88% (2397/2500) (classification)
- Compare to the previous experiment

Linear	Polynomial	RBF	Linear + RBF
95.92%	97.68%	98.52%	95.88%

e) GridSearch on linear combination of Linear kernel and RBF Kernel

Introduce a parameter **α** that make a linear combination of Linear kernel and RBF kernel. We have a new kernel function, defined as

α *Linear + **(1-α)**RBF kernel = K(x i , x j) = **α** Xi.T Xj + (1-**α)**exp(-γ||Xi - Xj || 2), γ > 0.

then do a grid-search to find a good setting of **α** and c

<code>

```
for log2c in range (-3,9,1):
  for alpha in range (1,10,2):
    LRBFKernelTrain = np.zeros((nTrain,nTrain))
    a = alpha*1.0/10
    b = 1 - a
      for i in range (nTrain):
        for j in range (i,nTrain,1):
          LRBFKernelTrain[i,j] = a*LinearKernelTrain[i][j+1] + b*RBFKernelTrain[i][j+1]
          LRBFKernelTrain[j,i] =  LRBFKernelTrain[i,j]

          RBFKernelTest = np.zeros((nTest,nTrain))

          for i in range (nTest):
            for j in range (1,nTrain,1):
              LRBFKernelTest[i,j] = a*LinearKernelTest[i][j+1] + b*LinearKernelTest[i][j+1]

    #add index to the first col
    index = range (1,nTrain+1)
    RBFKernelTrain = np.column_stack([index, LRBFKernelTrain])
    LRBFKernelTrain = (LRBFKernelTrain).tolist()

    #add index to the first col
    index = range (1,nTest+1)
    LRBFKernelTest = np.column_stack([index, LRBFKernelTest])
    LRBFKernelTest = (LRBFKernelTest).tolist()

    print (count,2**log2c,a,b )
    prob  = svm_problem(TrainLabel, LRBFKernelTrain, isKernel=True)
    param = svm_parameter("-q -t 4 -v 3 -c " + str(2**log2c))
    m = svm_train(prob, param)
    logSeting.append([2**log2c,a,b,m])
    if (m > bestCrossGrid) :
      bestCrossGrid = m
      bestc = 2**log2c
      besta= a
      bestb  = b
      count+=1
```

- The result of doing grid-search respect to **α** and c

c / α	0.125	0.25	0.5	1	2	4	8	16	32	64	128	256
0.1	96.98	**97.14**	97.08	96.9	96.48	96.72	96.68	96.58	96.5	96.28	96.44	96.78
0.3	96.86	96.94	96.66	96.54	96.28	96.26	96.48	96.24	96.5	96.46	96.42	96.3
0.5	96.82	96.72	96.44	96.18	96.18	96.42	96.2	96.42	96.3	95.92	96.4	96.56
0.7	96.82	96.38	96.42	96.28	96.22	96.18	96.38	96.22	96.2	96.44	96.22	96.08
0.9	96.76	96.26	96.42	96.42	96.5	96.04	96.08	96.18	96.1	96.16	96.24	96.16

The result is better than before combining 2 kennels equally, α = 0.1 and c = 0.25

The accuracy of test set then shows as

Accuracy = **96.08%** (2402/2500) (classification)

Linear	Polynomial	RBF	Linear + RBF	Linear + RBF refined
95.92%	97.68%	98.52%	95.88%	96.08%

- Using precomputed kernel give the result faster than using build-in kernel functions when we do the grid-search
- The RBF is the popular kernel that is used and the result of RBF is better than the others. When doing SVM, we need to consider about the computation time, since few experiment run on RBF and Polynomial are slower than normal.