VE445 Intro. to Machine Learning

REPORT OF
Lab 1
Graph

Name and ID
Sijun Zhang 516370910155

Professor: Bo Yuan

April 3, 2019

1 Introduction

In this experiment, the SMO algorithm is used to stimulate the hyper-plane and the goal function can be represented in the form of maximizing problem with constraint.

$$max \sum_{i=1}^{n} \alpha_i - 0.5 \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j x_i^T x_j$$

$$s.t. \quad \sum_{i=1}^{n} \alpha_i y_i = 0, \ \alpha_i \geq 0, \ i = 1, 2, 3...n$$

and when we use kernel to map the original sample vectors to higher dimension, and consider the soft margin problem the goal function becomes:

$$max \sum_{i=1}^{n} \alpha_i - 0.5 \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j K(x_i, x_j)$$

$$s.t. \quad \sum_{i=1}^{n} \alpha_i y_i = 0, \ C \geq \alpha_i \geq 0, \ i = 1, 2, 3...n$$

We also used the different prediction function under the kernel cases:

$$label = sign(\sum_{i=1}^{n} \alpha_i y_i K(x, x_i) + b)$$

Which means under the kernel case, we should use the whole training data set to predict the new input label, which is of course time-consuming and space-consuming.

2 Result

The train data is of 80% within the total sample data, and basically, we use the test data remained to test the accuracy of the svm model to demonstrate the performance of settable model. We used the SVC model in the library sklearn to serve as standard and the performance of the implemented SVM, the following recording is the mean of 3 same tests' results:

Result

data set	parameter$_b$	parameter$_w$	accuracy	skl$_w$	skl$_a cc$
1 in SVM	1.77	[0.92,1.86,2.80,3.67,4.59]	1.0	[1.00,2.00,2.99,3.99, 4.99]	
2 in kernelSVM	/	/	0.998	/	0.998
3 in softMarginSVM	2.74	[2.74,5.58,8.31,11.13,13.84]	0.999	[2.77,5.67,8.50,11.33,14.17]	0.998

In the data set 3, the SoftMarginSVM is set in (kernel = 'Linear', parameter = 0.1, C = 1.0), and in the data set 2, kernel = 'Gaussian', parameter = 0.1

And I also implement SoftMarginSVM on Iris data set

		Iris			
data set	parameter_b	parameter_w	accuracy	sklearn_w	sklearn_acc
Iris in softMarginSVM	7.023728	[-0.250741, -2.0059321]	0.9	[-0.234177, -2.0316551]	0.8

We can find there are still some offset between the sklearn.SVC and the implemented SMO, although the accuracy is high. When we compared the number of non-zero elements in α, in data set 1, we find the the implemented SMO has much more non-zero elements than expected. The implemented SMO algorithm is hard to converge to the sparse version, which results in the offset in the parameters.

3 Appendix

3.1 main.py

```
from SVM_SMO import SVM,KernelSVM,SoftMarginSVM
import csv
import numpy as np
import sklearn as sl
from sklearn import svm, datasets
from sklearn.utils import shuffle



if __name__ == '__main__':

    l1 = 'label_1.csv'
    s1 = 'sample_1.csv'
    with open(l1) as f:
        reader = csv.reader(f)
        ct_l1 = list(reader)
    ct_l1_np = np.array(ct_l1, dtype = float)
    with open(s1) as f:
        reader = csv.reader(f)
        ct_s1 = list(reader)
    ct_s1_np = np.array(ct_s1, dtype = float)

    ct_s1_np, ct_l1_np = shuffle(ct_s1_np, ct_l1_np, random_state = 10086)

    train_indices = np.random.choice(len(ct_s1_np),
                                int(round(len(ct_s1_np)*0.8)),
                                replace=False)
    test_indices = np.array(list(set(range(len(ct_s1_np))) - set(train_indices)))
    ct_s1_np_train = ct_s1_np[train_indices]
    ct_s1_np_test = ct_s1_np[test_indices]
    ct_l1_np_train = ct_l1_np[train_indices].astype(int)
    ct_l1_np_test = ct_l1_np[test_indices].astype(int)
```

```python
svm1 = SVM(ct_s1_np_train, ct_l1_np_train)
svm1.training()
print (svm1.parameter_w())
print (svm1.parameter_b())
print (svm1.print_sv_num())
acc1 = svm1.testing(ct_s1_np_test, ct_l1_np_test)
print (acc1)

M = svm.SVC(C = 100000000000000, kernel='linear')
M.fit(ct_s1_np, ct_l1_np)
print (M.coef_)
print (M.n_support_)


l2 = 'label_2.csv'
s2 = 'sample_2.csv'
with open(l2) as f:
    reader = csv.reader(f)
    ct_l2 = list(reader)
ct_l2_np = np.array(ct_l2, dtype = float)
with open(s2) as f:
    reader = csv.reader(f)
    ct_s2 = list(reader)
ct_s2_np = np.array(ct_s2, dtype = float)

ct_s2_np, ct_l2_np = shuffle(ct_s2_np, ct_l2_np, random_state = 10086)

train_indices = np.random.choice(len(ct_s2_np),
                             int(round(len(ct_s2_np)*0.8)),
                             replace=False)
test_indices = np.array(list(set(range(len(ct_s2_np))) - set(train_indices)))
ct_s2_np_train = ct_s2_np[train_indices]
ct_s2_np_test = ct_s2_np[test_indices]
ct_l2_np_train = ct_l2_np[train_indices].astype(int)
ct_l2_np_test = ct_l2_np[test_indices].astype(int)

print (len(np.where(ct_l2_np_test == 1) [0] ))

svm2 = KernelSVM(ct_s2_np_train, ct_l2_np_train)
svm2.training(kernel = 'Gaussian', parameter=0.1)
print (svm2.parameter_b())
print (svm2.print_sv_num())
acc2 = svm2.testing(ct_s2_np_test, ct_l2_np_test)
```

```
print (acc2)

M = svm.SVC(C = 100000000000, kernel='rbf', gamma=0.1)
M.fit(ct_s2_np_train, ct_l2_np_train)
print (M.n_support_)
print (M.score(ct_s2_np_test, ct_l2_np_test))



l3 = 'label_3.csv'
s3 = 'sample_3.csv'
with open(l3) as f:
    reader = csv.reader(f)
    ct_l3 = list(reader)
ct_l3_np = np.array(ct_l3, dtype = float)
with open(s3) as f:
    reader = csv.reader(f)
    ct_s3 = list(reader)
ct_s3_np = np.array(ct_s3, dtype = float)

ct_s3_np, ct_l3_np = shuffle(ct_s3_np, ct_l3_np, random_state = 10086)

train_indices = np.random.choice(len(ct_s3_np),
                             int(round(len(ct_s3_np)*0.8)),
                             replace=False)
test_indices = np.array(list(set(range(len(ct_s3_np))) - set(train_indices)))
ct_s3_np_train = ct_s3_np[train_indices]
ct_s3_np_test = ct_s3_np[test_indices]
ct_l3_np_train = ct_l3_np[train_indices].astype(int)
ct_l3_np_test = ct_l3_np[test_indices].astype(int)

svm3 = SoftMarginSVM(ct_s3_np_train, ct_l3_np_train)
svm3.training(kernel = 'Linear', parameter=0.1, C=1.0)
print (svm3.parameter_w())
print (svm3.parameter_b())
acc3 = svm3.testing(ct_s3_np_test, ct_l3_np_test)
print (acc3)

M = svm.SVC(C = 1.0, kernel='linear', gamma=0.1)
M.fit(ct_s3_np_train, ct_l3_np_train)
print (M.score(ct_s3_np_test, ct_l3_np_test))
print (M.coef_)

iris = datasets.load_iris()
ct_s4_np = iris.data[:, :2]
ct_l4_np = iris.target
```

```
    ct_l4_np = np.transpose(ct_l4_np)

    for i in range(len(ct_l4_np)):
        if (ct_l4_np[i] != 1):
            ct_l4_np[i] = -1

    ct_s4_np, ct_l4_np = shuffle(ct_s4_np, ct_l4_np, random_state = 10086)

    train_indices = np.random.choice(len(ct_s4_np),
                                 int(round(len(ct_s4_np)*0.8)),
                                 replace=False)
    test_indices = np.array(list(set(range(len(ct_s4_np))) - set(train_indices)))
    ct_s4_np_train = ct_s4_np[train_indices]
    ct_s4_np_test = ct_s4_np[test_indices]
    ct_l4_np_train = ct_l4_np[train_indices].astype(int)
    ct_l4_np_test = ct_l4_np[test_indices].astype(int)

    svm3 = SoftMarginSVM(ct_s4_np_train, ct_l4_np_train)
    svm3.training(kernel = 'Linear', parameter=0.1, C= 1.0)
    print (svm3.parameter_w())
    print (svm3.parameter_b())
    acc3 = svm3.testing(ct_s4_np_test, ct_l4_np_test)
    print (acc3)

    M = svm.SVC(C = 1.0, kernel='linear', gamma=0.1)
    M.fit(ct_s4_np_train, ct_l4_np_train)
    print (M.score(ct_s4_np_test, ct_l4_np_test))
    print (M.coef_)
```

3.2 SVMSMO.py

```
# This is the template of VE445 JI 2019 Spring Lab 1
# Name: Zhang, Sijun
#   ID: 516370910155
# Date: March 19th, 2019

import numpy as np
import random as rnd
import os
from numpy import *

MAX_FLOAT = 10000000.0
MAX_ITER = 20000
```

```
class SVM(object):
    # This class is the hard margin SVM and it is the parent
    # class of KernelSVM and SoftMarginSVM.
    # Please add any function to the class if it is needed.
    def __init__(self, sample, label):
    # This function is an constructor and shouldn't be modified.
    # The 'self.w' represents the director vector and should be
    # in the form of numpy.array
    # The 'self.b' is the displacement of the SVM and it should
    # be a float64 variable.
        self.dim = sample.shape[1]
        self.sample = sample
        self.label =label
        self.num = self.label.shape[0]
        self.a = np.zeros((self.num))
        self.w = np.zeros((self.dim))
        self.b = 0.0
        self.C = float ('inf')
        self.min_loss = 0.001
        self.max_iter = MAX_ITER
        self.kernel = 'Linear'
        self.parameter = 0.0

    def rand_index(self,l, h, i):
        j = i
        while j == i:
            j = rnd.randint(l,h-1)
        return j

    def training(self):
    # Implement this function by yourself and do not modifiy
    # the parameter.
        x = self.sample
        y = self.label
        y = (y.T[0])
        # self.kernel = kernel
        # self.parameter = parameter
        K = self.Linear(x, x)
        # print (K)
        ite = 0
        passes = 0
        max_passes = 3
        while ite < self.max_iter:
            num_changed_a = 0
            ite += 1
            print ('ite  = '+ str(ite))
```

```
for i in range (self.num):
    t = (self.a * y)
    Ei = np.dot(t, K[i] )+ self.b - y[i]
    if (y[i]*Ei < -self.min_loss and self.a[i] < self.C) or (y[i]*Ei > self.min_loss and
        j = self.rand_index(0, self.num, i)
        Ej = np.dot(t, K[j] )+ self.b - y[j]
        aio = self.a[i]
        ajo = self.a[j]
        if y[i] != y[j]:
            L = max(0, self.a[j] - self.a[i])
            H = MAX_FLOAT
        else:
            L = 0
            H = self.a[i] + self.a[j]
        if L==H:
            continue
        eta = 2*K[i, j] - K[i, i] - K[j, j]
        if eta >= 0:
            continue
        ajn = self.a[j] - y[j]*(Ei-Ej)/eta
        if ajn > H:
            self.a[j] = H
        elif ajn < L:
            self.a[j] = L
        else:
            self.a[j] = ajn
        if (self.a[j] - ajo < 0.00001 and self.a[j] - ajo > -0.00001):
            continue
        self.a[i] = self.a[i] + y[i]*y[j]*(ajo-self.a[j])

        #b part
        b1 = self.b - Ei - y[i]*(self.a[i] - aio)*K[i,i] - y[j]*(self.a[j] - ajo) * K[i,
        b2 = self.b - Ei - y[i]*(self.a[i] - aio)*K[i,j] - y[j]*(self.a[j] - ajo) * K[j,

        if (self.a[i] > 0 and self.a[i] < self.C):
            self.b = b1
        elif (self.a[j] > 0 and self.a[j] < self.C):
            self.b = b2
        else:
            self.b = (b1 + b2) /2.0
        num_changed_a = num_changed_a + 1
if num_changed_a <= 3:
    passes += 1
else:
    passes = 0
print ('num_changed_a = ' + str(num_changed_a))
```

```
        if (passes >= max_passes):
            break
    self.w = np.dot(x.T, np.multiply(self.a, y))


def Linear(self, x, y):
    return np.matmul(x, y.T)

def testing(self, test_sample, test_label):
# This function should return the accuracy
# of the input test_sample in float64, e.g 0.932
    y_vat = np.sign(np.dot(self.w.T, test_sample.T) +self.b).astype(int)
    y = (np.transpose(test_label)[0])
    # print (y)
    idx = np.where(y_vat == 1)
    tp = np.sum( abs( y[idx] - y_vat[idx] < 0.0001) )
    idx = np.where(y_vat == -1)

    tn = np.sum( abs( y[idx] - y_vat[idx] < 0.0001))

    return float(tp + tn ) / len( y)

def print_sv_num(self):
    cnt = [0,0]
    for i in range(len(self.a)):
        if (self.a[i] != 0) & (self.label[i] == 1):
            cnt[0] += 1
        elif (self.a[i] != 0) & (self.label[i] == -1):
            cnt[1] +=1
    return cnt


def parameter_w(self):
# This function is used to return the parameter w of the SVM.
# The result is supposed to be an np.array
# This functin shouldn't be modified.
    return self.w
def parameter_b(self):
# This function is used to return the parameter self.b of the SVM.
# The result is supposed to be an real number.
# This functin shouldn't be modified.
    return self.b

# If you choose to use tf.InteractiveSession, please remember
# to close it or there might be memory overflow.
# You can recycle the resource by using a destructor.
```

```
class KernelSVM(SVM):
    # This class is the kernel SVM.
    # Please add any function to the class if it is needed.

    def training(self, kernel = 'Linear', parameter = 1):
    # Specifics:
    #   For the parameter of 'kernel':
    #   1. The default kernel function is 'Linear'.
    #      The parameter is 1 by default.
    #   2. Gaussian kernel function is 'Gaussian'.
    #      The parameter is the Gaussian bandwidth.
    #   3. Laplace kernel funciton is 'Laplace'.
    #   4. Polynomial kernel functino is 'Polynomial'.
    #      The parameter is the exponential of polynomial.
    # Add your cold after the initialization.
        self.kernel = kernel
        self.parameter = parameter
        x = self.sample
        y = self.label
        y = (y.T[0])
        # self.kernel = kernel
        # self.parameter = parameter
        if self.kernel == 'Linear':
            K = self.Linear(x,x)
        elif self.kernel == 'Polynomial':
            K = self.Polynomial(x,x, self.parameter)
        elif self.kernel == 'Gaussian':
            K = self.Gaussian(x,x,self.parameter)
        elif self.kernel == 'Laplace':
            K = self.Laplace(x,x, self.parameter)

        # print (K)
        ite = 0
        passes = 0
        max_passes = 3
        while ite < self.max_iter:
            num_changed_a = 0
            ite += 1
            print ('ite  = '+ str(ite))
            for i in range (self.num):
                t = (self.a * y)
                # print (t)
                # print (y.shape)
                Ei = np.dot(t, K[i] )+ self.b - y[i]
```

```
    # print (y[i])
    # print (Ei)
    # print (self.a[i])
    if (y[i]*Ei < -self.min_loss and self.a[i] < self.C) or (y[i]*Ei > self.min_loss and
        j = self.rand_index(0, self.num, i)
        Ej = np.dot(t, K[j] )+ self.b - y[j]
        aio = self.a[i]
        ajo = self.a[j]
        if y[i] != y[j]:
            L = max(0, self.a[j] - self.a[i])
            H = MAX_FLOAT
        else:
            L = 0
            H = self.a[i] + self.a[j]
        if L==H:
            continue
        eta = 2*K[i, j] - K[i, i] - K[j, j]
        if eta >= 0:
            continue
        ajn = self.a[j] - y[j]*(Ei-Ej)/eta
        if ajn > H:
            self.a[j] = H
        elif ajn < L:
            self.a[j] = L
        else:
            self.a[j] = ajn
        if (self.a[j] - ajo < 0.00001 and self.a[j] - ajo > -0.00001):
            continue
        self.a[i] = self.a[i] + y[i]*y[j]*(ajo-self.a[j])

        #b part
        b1 = self.b - Ei - y[i]*(self.a[i] - aio)*K[i,i] - y[j]*(self.a[j] - ajo) * K[i,
        b2 = self.b - Ei - y[i]*(self.a[i] - aio)*K[i,j] - y[j]*(self.a[j] - ajo) * K[j,

        if (self.a[i] > 0 and self.a[i] < self.C):
            self.b = b1
        elif (self.a[j] > 0 and self.a[j] < self.C):
            self.b = b2
        else:
            self.b = (b1 + b2) /2.0
        num_changed_a = num_changed_a + 1
if num_changed_a <= 3:
    passes += 1
else:
    passes = 0
print ('num_changed_a = ' + str(num_changed_a))
```

```
            if (passes >= max_passes):
                break
        if (self.kernel == 'Linear'):
            self.w = np.dot(x.T, np.multiply(self.a, y))


    def testing(self, test_sample, test_label):
        y = test_label
        if (len(y) >=1000 ):
            y = (y.T[0])
        y_train = np.transpose(self.label)[0]
        if self.kernel == 'Linear':
            y_vat = np.sign(np.dot(self.w.T, test_sample.T) +self.b).astype(int)
        elif self.kernel == 'Polynomial':
            y_vat = np.sign(np.add(np.matmul(np.multiply(self.a, y_train), self.Polynomial(self.samp
        elif self.kernel == 'Gaussian':
            rA = np.reshape(np.square(self.sample).sum(axis = 1),[-1,1])
            rB = np.reshape(np.square(test_sample).sum(axis = 1),[-1,1])
            print ('in gaussian predict')
            sq_dists = np.sqrt(np.abs(np.add(np.subtract(rA, np.multiply(2, np.matmul(self.sample, n
            % print (sq_dists)
            temp = np.exp(-sq_dists/(2*pow(self.parameter, 2)))
            % print (temp)
            o = np.add(np.matmul(np.multiply(self.a, y_train), temp), self.b)
            y_vat = np.sign(o).astype(int)
        elif self.kernel == 'Laplace':
            gamma = self.parameter
            rA = np.reshape(np.square(self.sample).sum(axis = 1),[-1,1])
            rB = np.reshape(np.square(test_sample).sum(axis = 1),[-1,1])
            sq_dists = np.sqrt(np.abs(np.add(np.subtract(rA, np.multiply(2, np.matmul(self.sample, n
            temp = np.exp(-sq_dists/gamma)
            y_vat = np.sign(np.add(np.matmul(np.multiply(self.a, y_train), temp), self.b)).astype(in

        print (len (np.where(y_vat == 1)[0]) )
        print (y)
        print (y_vat)

        idx = np.where(y_vat == 1)
        tp = np.sum( abs( y[idx] - y_vat[idx] < 0.0001) )
        idx = np.where(y_vat == -1)

        tn = np.sum( abs( y[idx] - y_vat[idx] < 0.0001))

        return float(tp + tn ) / len( y)
```

```
    def Polynomial(self, x, y, d):
        return pow(np.matmul(x,y.T),d )
    def Gaussian(self, x, y, gamma):
        dist = np.reshape(np.square(x).sum(axis = 1), [-1,1] )
        sq_d = np.sqrt(np.abs(np.add(np.subtract(dist, np.multiply(2, np.matmul(x, y.T))), np.transp
        return (-sq_d/(2*(gamma**2)))
    def Laplace(self, x, y, gamma):
        dist = np.reshape(np.square(x).sum(axis = 1), [-1,1] )
        sq_d = np.sqrt(np.abs(np.add(np.subtract(dist, np.multiply(2, np.matmul(x, y.T))), np.transp
        return (-sq_d/gamma)

class SoftMarginSVM(KernelSVM):
    # This class is the soft margin SVM and inherits
    # the kernel SVM to expand to both linear Non-seperable and
    # soft margin problem.
    # Please add any function to the class if it is needed.
    def training(self, kernel = 'Linear', parameter = 1, C = 1.0):
    # Specifics:
    #   For the parameter of 'kernel':
    #   1. The default kernel function is 'Linear'.
    #      The parameter is 1 by default.
    #   2. Gaussian kernel function is 'Gaussian'.
    #      The parameter is the Gaussian bandwidth.
    #   3. Laplace kernel funciton is 'Laplace'.
    #   4. Polynomial kernel functino is 'Polynomial'.
    #      The parameter is the exponential of polynomial.
    # Add your cold after the initialization.
        self.C = C
        self.kernel = kernel
        self.parameter = parameter
        x = self.sample
        y = self.label
        if (len(y) >=1000 ):
            y = (y.T[0])
        # self.kernel = kernel
        # self.parameter = parameter
        if self.kernel == 'Linear':
            K = self.Linear(x,x)
        elif self.kernel == 'Polynomial':
            K = self.Polynomial(x,x, self.parameter)
        elif self.kernel == 'Gaussian':
            K = self.Gaussian(x,x,self.parameter)
        elif self.kernel == 'Laplace':
            K = self.Laplace(x,x, self.parameter)

        # print (K)
```

```
ite = 0
passes = 0
max_passes = 3
while ite < self.max_iter:
    num_changed_a = 0
    ite += 1
    print ('ite  = '+ str(ite))
    for i in range (self.num):
        t = (self.a * y)
        # print (t)
        # print (y.shape)
        Ei = np.dot(t, K[i] )+ self.b - y[i]
        # print (y[i])
        # print (Ei)
        # print (self.a[i])
        if (y[i]*Ei < -self.min_loss and self.a[i] < self.C) or (y[i]*Ei > self.min_loss and
            j = self.rand_index(0, self.num, i)
            Ej = np.dot(t, K[j] )+ self.b - y[j]
            aio = self.a[i]
            ajo = self.a[j]
            if y[i] != y[j]:
                L = max(0, self.a[j] - self.a[i])
                H = min(C, C + self.a[j] - self.a[i])
            else:
                L = max(0, self.a[i] + self.a[j] - C)
                H = min(C, self.a[i] + self.a[j])
            if L==H:
                continue
            eta = 2*K[i, j] - K[i, i] - K[j, j]
            if eta >= 0:
                continue
            ajn = self.a[j] - y[j]*(Ei-Ej)/eta
            if ajn > H:
                self.a[j] = H
            elif ajn < L:
                self.a[j] = L
            else:
                self.a[j] = ajn
            if (self.a[j] - ajo < 0.00001 and self.a[j] - ajo > -0.00001):
                continue
            self.a[i] = self.a[i] + y[i]*y[j]*(ajo-self.a[j])

            #b part
            b1 = self.b - Ei - y[i]*(self.a[i] - aio)*K[i,i] - y[j]*(self.a[j] - ajo) * K[i,
            b2 = self.b - Ei - y[i]*(self.a[i] - aio)*K[i,j] - y[j]*(self.a[j] - ajo) * K[j,
```

```
            if (self.a[i] > 0 and self.a[i] < self.C):
                self.b = b1
            elif (self.a[j] > 0 and self.a[j] < self.C):
                self.b = b2
            else:
                self.b = (b1 + b2) /2.0
            num_changed_a = num_changed_a + 1
    if num_changed_a <= 3:
        passes += 1
    else:
        passes = 0
    print ('num_changed_a = ' + str(num_changed_a))
    if (passes >= max_passes):
        break
if (self.kernel == 'Linear'):
    self.w = np.dot(x.T, np.multiply(self.a, y))
```