task2

April 20, 2023

1 Task 2

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import math
import re
%config InlineBackend.figure_format = 'retina'
%matplotlib inline
import scipy.io as sio
plt.rcParams['figure.figsize'] = 10,10

import sklearn.datasets
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import accuracy_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
```

1.1 Original Data

[ ]:
```
my_iris_data = sklearn.datasets.load_iris()
print ("my_iris_data.data.shape:",my_iris_data.data.shape)
print ("labels:",my_iris_data.target)
```

```
my_iris_data.data.shape: (150, 4)
labels: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

[ ]:
```
X_train = np.concatenate((my_iris_data.data[10:50,:],my_iris_data.data[60:100,:
↪], my_iris_data.data[110:150,:]))
X_train = np.concatenate((np.ones((X_train.shape[0],1)),X_train),axis=1) #␣
↪Append bias term 1
```

```
y_train = np.concatenate((my_iris_data.target[10:50],my_iris_data.target[60:
 ↪100], my_iris_data.target[110:150]))
print ("X_train.shape:", X_train.shape)
print ("y_train.shape:", y_train.shape)

X_test = np.concatenate((my_iris_data.data[40:50,:],my_iris_data.data[90:100,:
 ↪], my_iris_data.data[140:150,:]))
X_test = np.concatenate((np.ones((X_test.shape[0],1)),X_test),axis=1) # Append␣
 ↪bias term 1
y_test = np.concatenate((my_iris_data.target[40:50],my_iris_data.target[90:
 ↪100], my_iris_data.target[140:150]))
print ("X_test.shape:", X_test.shape)
print ("y_test.shape:", y_test.shape)
```

```
X_train.shape: (120, 5)
y_train.shape: (120,)
X_test.shape: (30, 5)
y_test.shape: (30,)
```

1.2 Preprocess the labels to get 3 datasets

[ ]:
```
y_train1 = np.copy(y_train); y_test1 = np.copy(y_test)
y_train2 = np.copy(y_train); y_test2 = np.copy(y_test)
y_train3 = np.copy(y_train); y_test3 = np.copy(y_test)

y_train1[y_train == 1] = -1
y_train1[y_train == 2] = -1
y_train1[y_train == 0] = 1
y_test1[y_test == 1] = -1
y_test1[y_test == 2] = -1
y_test1[y_test == 0] = 1

y_train2[y_train == 1] = 1
y_train2[y_train == 2] = -1
y_train2[y_train == 0] = -1
y_test2[y_test == 1] = 1
y_test2[y_test == 2] = -1
y_test2[y_test == 0] = -1

y_train3[y_train == 1] = -1
y_train3[y_train == 2] = 1
y_train3[y_train == 0] = -1
y_test3[y_test == 1] = -1
y_test3[y_test == 2] = 1
y_test3[y_test == 0] = -1
```

1.3 Task2: Explicit SVM on iris

Original target function:

$$L(\mathbf{w}_1, \dots, \mathbf{w}_K) = \frac{1}{2}\sum_{k=1}^{K} \|\mathbf{w}_k\|^2 + C\sum_{i}\sum_{k=1,k\neq y_i}^{K} max(0, 1 - (<\mathbf{w}_{y_i}, \mathbf{x}_i> - <\mathbf{w}_k, \mathbf{x}_i>))$$

The gradient w.r.t $\mathbf{w}$ of the target function:

$$L'(\mathbf{w}_b) = \frac{dL(\mathbf{w}_b)}{d\mathbf{w}_b} = \mathbf{w}_b + C\sum_{i}\sum_{k=1,k\neq y_i}^{K} \begin{cases} \mathbf{x}_i & , \texttt{ if } (b = k) \wedge (b \neq y_i) \wedge (<\mathbf{w}_{y_i}, \mathbf{x}_i> - <\mathbf{w}_k, \mathbf{x}_i> < 1) \\ -\mathbf{x}_i & , \texttt{ if } (b \neq k) \wedge (b = y_i) \wedge (<\mathbf{w}_{y_i}, \mathbf{x}_i> - <\mathbf{w}_k, \mathbf{x}_i> < 1) \\ 0 & , \texttt{ otherwise} \end{cases}$$

[ ]:
```
C=2 # the lambda
learning_rate = 0.0001 # the alpha
n_iter = 20000
iterations = []
```

[ ]:
```
# gradient of loss function L(w)
def L_prime_w(X, Y, ws):
    grad = np.zeros((X.shape[1],0))
    for b in range(ws.shape[1]):
        sum_loss = np.zeros((X.shape[1]))
        for i in range(X.shape[0]):
            for k in range(ws.shape[1]):
                if k != Y[i]:
                    dot1 = np.dot(ws[:,Y[i]], X[i])
                    dot2 = np.dot(ws[:,k], X[i])
                    if (b == k) and (b != Y[i]) and (dot1 - dot2 < 1):
                        sum_loss += X[i]
                    elif (b != k) and (b == Y[i]) and (dot1 - dot2 < 1):
                        sum_loss += -X[i]
        grad = np.hstack([grad, (ws[:,b]+C*sum_loss).reshape(-1, 1)])
    return grad
```

[ ]:
```
def L_w(X, Y, ws):
    loss_sum = 0
    l2_sum = 0
    for i in range(len(X)):
        for k in range(ws.shape[1]):
            if k != Y[i]:
                dot1 = np.dot(ws[:,Y[i]], X[i])
                dot2 = np.dot(ws[:,k], X[i])
                hinge_loss = np.maximum(0, 1 - (dot1 - dot2))
                loss_sum += hinge_loss
    for k in range(ws.shape[1]):
```

```
        l2 = np.dot(ws[k].T, ws[k])
        l2_sum += l2
    ret = 0.5 * l2_sum + C * loss_sum
    return ret
```

1.3.1 (Warning! The next cell takes time to finish descending!)

[ ]:
```
w = np.zeros((X_train.shape[1], 3))

# We will keep track of training loss over iterations
iterations = [0]
L_w_list = [L_w(X_train, y_train, w)]
for i in range(n_iter):

    gradient = L_prime_w(X_train, y_train, w)
    # print(gradient)
    w_new = w - learning_rate * gradient
    iterations.append(i+1)
    L_w_list.append(L_w(X_train, y_train, w_new))

    if np.linalg.norm(w_new - w, ord = 1) < 0.001:
        print("gradient descent has converged after " + str(i) + " iterations")
        break
    if i % 1000 == 0:
        print(i, np.linalg.norm(w_new - w, ord = 1), L_w_list[-1])
    w = w_new

print ("w vector: \n" + str(w))
```

```
0 0.10575999999999999 413.4471309231998
1000 0.006433916072793366 33.53332247533323
2000 0.012583001565076524 29.332230312220247
gradient descent has converged after 2136 iterations
w vector:
[[ 0.24049589  0.61570382 -0.85619971]
 [ 0.54962307  0.34553133 -0.8951544 ]
 [ 0.94533576  0.151773   -1.09710876]
 [-1.40369287 -0.18671239  1.59040526]
 [-0.77774176 -0.88098094  1.6587227 ]]
```

1.3.2 Results on Training set

[ ]:
```
prediction = np.argmax(np.dot(X_train, w),axis=1)
training_accuracy = (prediction - y_train == 0)

print ("The training accuracy:", np.sum(training_accuracy)*1.0/X_train.
 ↪shape[0]*100, "%.")
```

The training accuracy: 97.5 %.

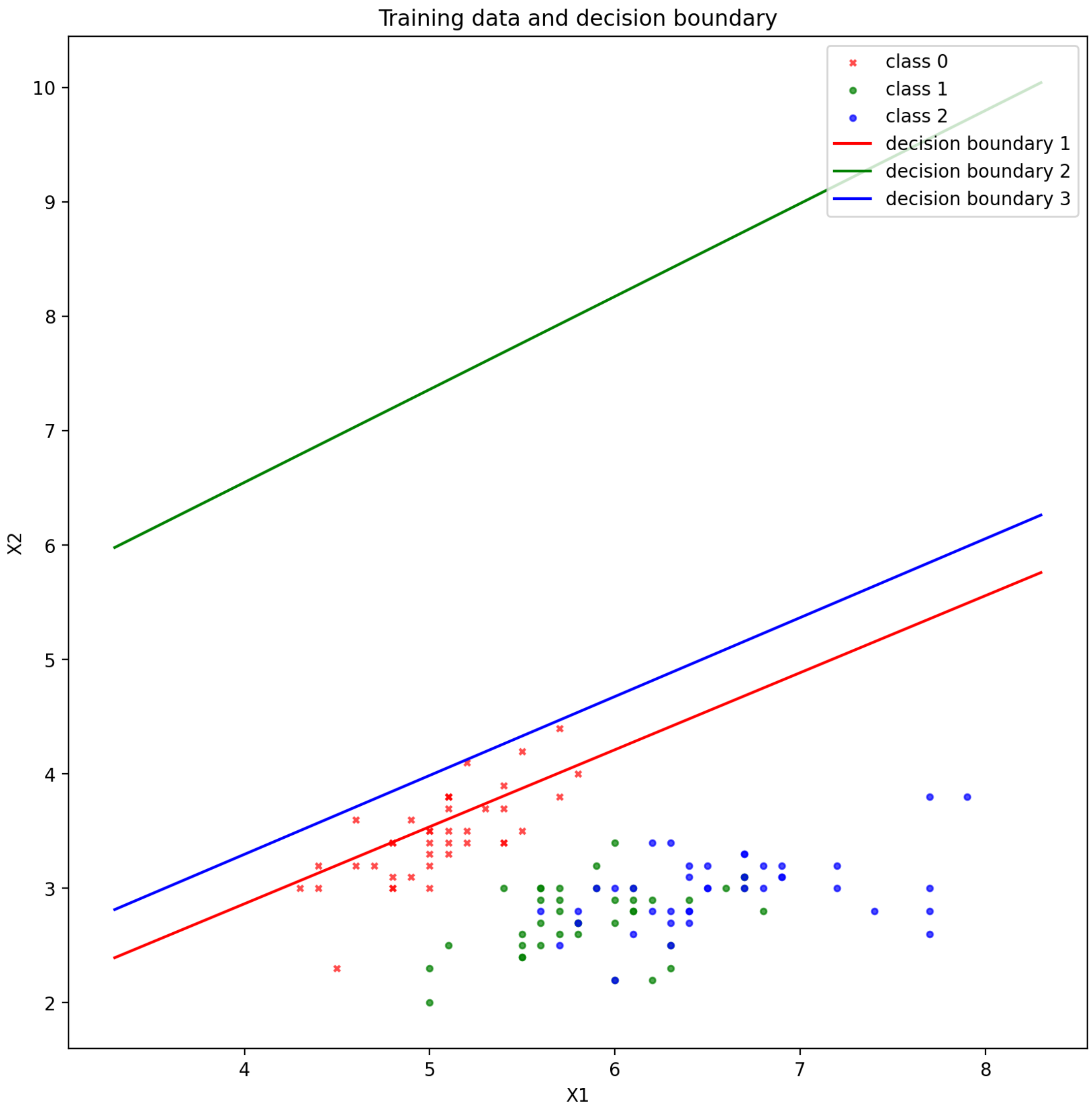
[ ]:
```
x1 = 1; x2 = 2
x = np.arange(np.min(X_train[:,x1])-1,np.max(X_train[:,x1])+1,1.0)
y1 = (-w[0][0]-w[2][0]*x)/w[3][0]
y2 = (-w[0][1]-w[2][1]*x)/w[3][1]
y3 = (-w[0][2]-w[2][2]*x)/w[3][2]

plt.scatter(X_train[y_train==0, x1], X_train[y_train==0, x2], marker='x',␣
↪color='r', alpha=0.7, s=10, label='class 0')
plt.scatter(X_train[y_train==1, x1], X_train[y_train==1, x2], marker='o',␣
↪color='g', alpha=0.7, s=10, label='class 1')
plt.scatter(X_train[y_train==2, x1], X_train[y_train==2, x2], marker='o',␣
↪color='b', alpha=0.7, s=10, label='class 2')

plt.xlabel('X1')
plt.ylabel('X2')
plt.plot(x,y1, color='r', label='decision boundary 1')
plt.plot(x,y2, color='g', label='decision boundary 2')
plt.plot(x,y3, color='b', label='decision boundary 3')
plt.title('Training data and decision boundary')

plt.legend(loc='upper right', fontsize=10)
```

[ ]: <matplotlib.legend.Legend at 0x7fb450106da0>



1.3.3 Results on Test set

[ ]:
```
#prediction = 2 * (np.dot(X_test, w) >= 0) - 1
#prediction = sigmoid(np.dot(X_test, w)) >= 0.5
prediction = np.argmax(np.dot(X_test, w), axis=1)

testing_accuracy = np.sum(prediction == y_test)*1.0/X_test.shape[0]
print ("The test accuracy: ", testing_accuracy*100, "%.")
```

```
The test accuracy:  100.0 %.
```