Homework 1 Report

Vivian Cheung

April 2023

1 Task 1

For task 1, we had to implement on-vs-all SVM using gradient descent. Our loss function was:

$$L(\mathbf{w}) = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_i max(0, 1 - y_i \times < \mathbf{x}_i, \mathbf{w} >) \tag{1}$$

and our gradient with respect to w of the target function was:

$$L'(\mathbf{w}) = \frac{dL(\mathbf{w})}{d\mathbf{w}} = \mathbf{w} + C\sum_i \begin{cases} -y_i\mathbf{x}_i & \texttt{, if } \ y_i \times < \mathbf{x}_i, \mathbf{w} > \leq 1 \\ 0 & \texttt{, otherwise} \end{cases} \tag{2}$$

After writing out the L_prime_w and L_w functions, we trained and tested with C=10 and a learning rate of 0.0001. With C=10, our training accuracy is 100%, 77.5%, and 95.83% for data sets 1, 2, and 3, respectively. The total training accuracy was 95.83%

The resulting plot with the three decision boundaries was interesting. What is seen with the red and green boundaries is expected since this is a one-vs-all implementation, so they do a fairly good job of separating the classes. As class 1 and class 2 are mixed within each other, it is understandable why decision boundary 2 mainly separates the red from everything else. However, what is interesting is decision boundary 3, which is far above all of the data points and does not separate the blue from all the other points.

Overall, the resulting plot is interesting to look at given that this still has a training accuracy of 95.83% and a test accuracy of 100%.

2 Task 2

For task 2, we implemented an explicit multi-class SVM that uses multiple hyperplanes for classification. Our loss function was:

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

and our gradient w.r.t. w of the loss function was:

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

After writing out the L_prime_w and L_w functions, we train and test our training sets. Our result shows that it converges after 2136 iterations with our loss decreasing significantly after the first iteration.

In our results on the training set, we have a training accuracy of 97.5%. This is surprisingly high given what is seen in the plot. There, it seems like only decision boundary 1 is remotely close to the correct classification, while decision boundaries 2 and 3 are much further from the rest of the data points, correct or not. The resulting test accuracy is 100

3 Task 3

Similar to the previous two tasks, in task 3, we implement a softmax classifier to classify the data set. Our loss function is:

$$L(\mathbf{w}_1, \ldots, \mathbf{w}_K, b_1, \ldots, b_K) = -\sum_i \ln p_{y^{(i)}} + \frac{\lambda}{2} \sum_{k=1}^{K} \|\mathbf{w}_k\|^2 .$$
$$\text{where } p_j = p(y = j|\mathbf{x}) = \frac{e^{f_j}}{\sum_{k=1}^{K} e^{f_k}}; f_j = \mathbf{w}_j \cdot \mathbf{x} + b_j$$

The gradient of the loss function is:

$$\frac{dL(\mathbf{w}_1, \ldots, \mathbf{w}_K, b_1, \ldots, b_K)}{d\mathbf{w}_k} = \lambda \mathbf{w}_k + \sum_{i, y_i = k} (p(k|\mathbf{x}_i) - 1)\mathbf{x}_i + \sum_{i, y_i \neq k} p(k|\mathbf{x}_i)\mathbf{x}_i. \quad (3)$$

$$\frac{dL(\mathbf{w}_1, \ldots, \mathbf{w}_K, b_1, \ldots, b_K)}{db_k} = \sum_{i, y_i = k} (p(k|\mathbf{x}_i) - 1) + \sum_{i, y_i \neq k} p(k|\mathbf{x}_i). \quad (4)$$

In our implementation of the L_prime_w and L_w functions, we use the provided softmax function to find the probabilities of class k. In our results, we see that the loss decreases all the way to 20 after 5000 iterations. The resulting training accuracy is 97.5%, and the plot shows a reasonable result given softmax classifier. As it normalizes the probabilities of classifications. Therefore, relative to the distribution of the data points, it is justified why the decision boundaries 1 and 3 are below the data points, while decision boundary 2 is above the data points.

Overall, we see a test set accuracy of 100%, similar to the results in tasks 1 and 2.

task1

April 20, 2023

1 Task 1

[ ]:
```
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 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]
```

[ ]:
```
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 Task 1: OvA SVM on iris

We use gradient descent to train 3 SVMs

Original target function:

$$L(\mathbf{w}) = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_i max(0, 1 - y_i \times < \mathbf{x}_i, \mathbf{w} >)$$

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

$$L'(\mathbf{w}) = \frac{dL(\mathbf{w})}{d\mathbf{w}} = \mathbf{w} + C\sum_i \begin{cases} -y_i\mathbf{x}_i & \text{, if } y_i \times < \mathbf{x}_i, \mathbf{w} > \leq 1 \\ 0 & \text{, otherwise} \end{cases}$$

[ ]:
```
C=10 # the lambda
learning_rate = 0.0001 # the alpha
n_iter = 20000
iterations = []
```

[ ]:
```
# gradient of loss function L(w)
def L_prime_w(X, Y, w):
    grad = np.zeros((X.shape[1]))
    for i in range(len(X)):
        if Y[i] * (np.dot(X[i], w).reshape(-1, 1)) <= 1:
            grad += (-Y[i]*X[i])
    grad = w + C * grad.reshape(-1, 1)
    return grad
```

[ ]:
```
def L_w(X, Y, w):
    ret = np.zeros((X.shape[1],1))
    for i in range(len(X)):
        hinge_loss = np.maximum(0, 1 - np.dot(X[i], w).reshape(-1, 1) * Y[i])
        ret += hinge_loss
    ret = 0.5 * np.dot(w.T, w) + C * ret
    return ret
```

[ ]:
```
#w = np.random.randn(X_train.shape[1], 1)
def train_svm(X_train, Y_train):
    w = np.zeros((X_train.shape[1],1))

    for i in range(n_iter):
        gradient = L_prime_w(X_train, Y_train, w)
        w_new = w - learning_rate * gradient
        iterations.append(i+1)

        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))
        w = w_new
    return w
```

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

[ ]:
```
w1 = np.copy(train_svm(X_train, y_train1))
print ("w1 vector:",  w1.tolist())
w2 = np.copy(train_svm(X_train, y_train2))
print ("w2 vector:",  w2.tolist())
w3 = np.copy(train_svm(X_train, y_train3))
print ("w3 vector:",  w3.tolist())
```

```
0 0.8824999999999998
gradient descent has converged after 165 iterations
w1 vector: [[0.13706870445168154], [0.22387327285027772], [0.6217347797476844],
[-1.0914360336784867], [-0.51157248524306]]
0 0.5651
1000 0.4609553624356504
2000 0.36728835264384024
3000 0.41975014563197854
4000 0.43329862753334714
5000 0.43223791544773627
6000 0.43226824464687863
7000 0.3592903156904593
8000 0.4026155586485771
9000 0.42174010320938327
10000 0.4026564584524035
11000 0.4217745861825237
12000 0.4026896359188892
13000 0.40270111318761326
14000 0.43301172706941754
15000 0.35941377794509227
16000 0.3703171483767201
17000 0.3703169861551151
18000 0.37031689739481044
19000 0.37031726981108254
w2 vector: [[6.391768255010402], [-0.31933682516848694], [-2.2502337671068617],
[1.6890043991911978], [-3.304710022048254]]
0 0.3689000000000001
1000 0.01895838433401842
2000 0.0031263054072552254
3000 0.053403942484862243
```

```
4000 0.027836688344175187
5000 0.027744036668115424
6000 0.027878118095105764
7000 0.009915442457714363
8000 0.0278980728760696
9000 0.006805368397893252
10000 0.009888375509008895
11000 0.006819595877694429
12000 0.00682460320278544
13000 0.00987095655623249
14000 0.027935303505910003
15000 0.009861598831906071
16000 0.0098572882953889
17000 0.027653199031345665
18000 0.006852837278106705
19000 0.027958323026417098
w3 vector: [[-4.407285012069442], [-0.9279616859573689], [-1.966196388583646],
[2.0005039147825423], [3.703731967413868]]
```

[ ]:
```
w= np.concatenate((w1,w2,w3),axis=1)
print ("w.shape:", w.shape)
```

```
w.shape: (5, 3)
```

[ ]:
```
def eva_accuracy(X_train, y_train, w):
    prediction = 2 * (np.dot(X_train, w) >= 0) - 1
    accuracy = np.sum(prediction == y_train.reshape(-1, 1))*1.0/X_train.shape[0]
    return accuracy
# print(prediction.shape, Y_test.shape)
accuracy1 = eva_accuracy(X_train, y_train1, w1)
print ("training accuracy1: " + str(accuracy1))
accuracy2 = eva_accuracy(X_train, y_train2, w2)
print ("training accuracy2: " + str(accuracy2))
accuracy3 = eva_accuracy(X_train, y_train3, w3)
print ("training accuracy3: " + str(accuracy3))
```

```
training accuracy1: 1.0
training accuracy2: 0.775
training accuracy3: 0.9583333333333334
```

1.3.2 Results on training set

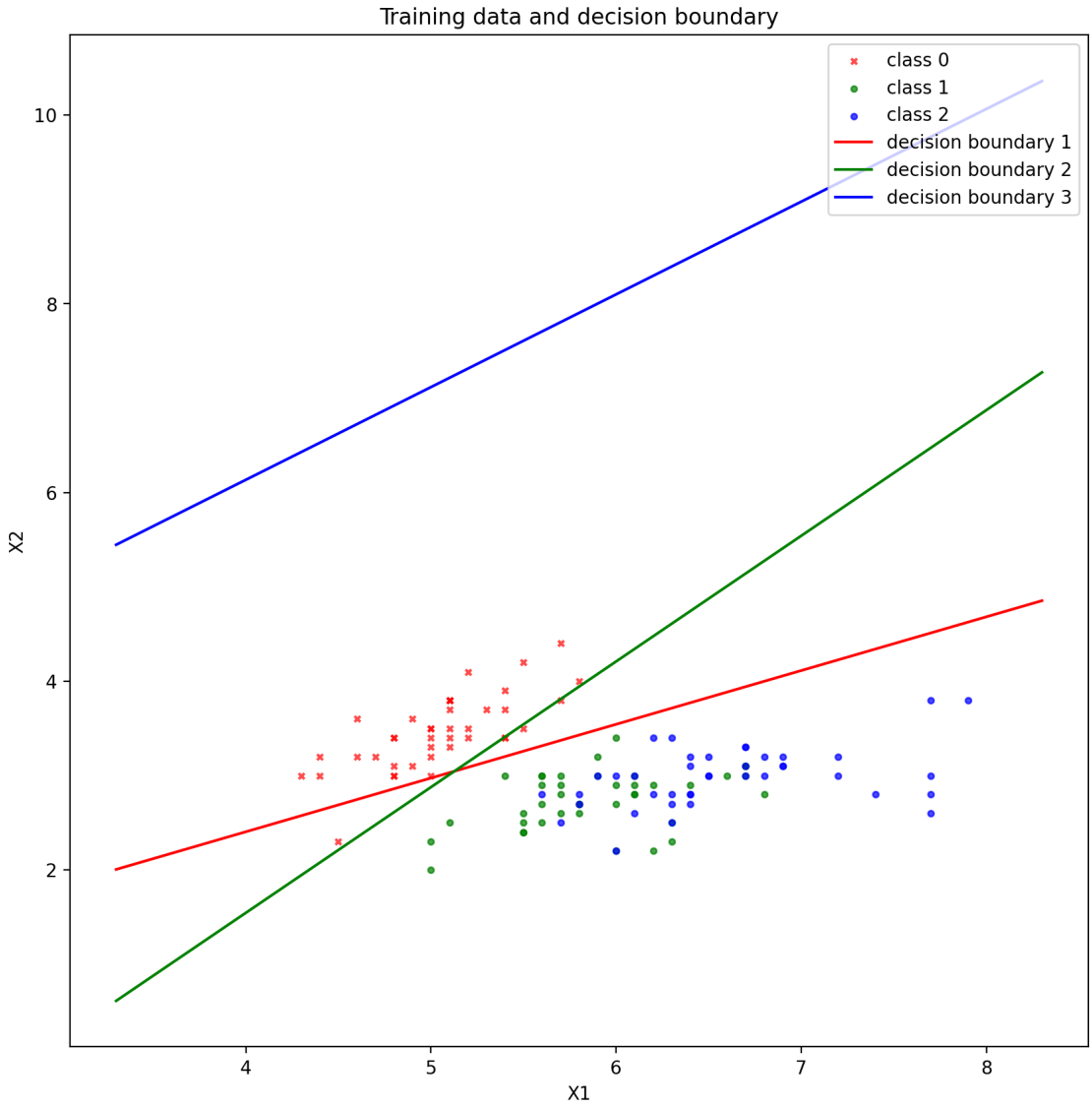
[ ]:
```
prediction1 = np.dot(X_train, w1)
prediction2 = np.dot(X_train, w2)
prediction3 = np.dot(X_train, w3)
preds = np.concatenate((prediction1, prediction2, prediction3),axis=1)
pred = np.argmax(preds, axis=1)
```

```
total_accuracy = np.sum(pred == y_train)*1.0/X_train.shape[0]
print ("Total training accuracy:", total_accuracy*100, "%.")
```

```
Total training accuracy: 95.83333333333334 %.
```

[ ]:
```
x1 = 1; x2 = 2
x = np.arange(np.min(X_train[:,1])-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 0x7f787086a5f0>
```



1.3.3 Results on test set

[ ]:
```
prediction1 = np.dot(X_test, w1)
prediction2 = np.dot(X_test, w2)
prediction3 = np.dot(X_test, w3)
preds = np.concatenate((prediction1, prediction2, prediction3),axis=1)
pred = np.argmax(preds, axis=1)

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

```
Total test accuracy: 100.0 %.
```

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 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 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
 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:

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

[ ]:
```
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 ]]
```

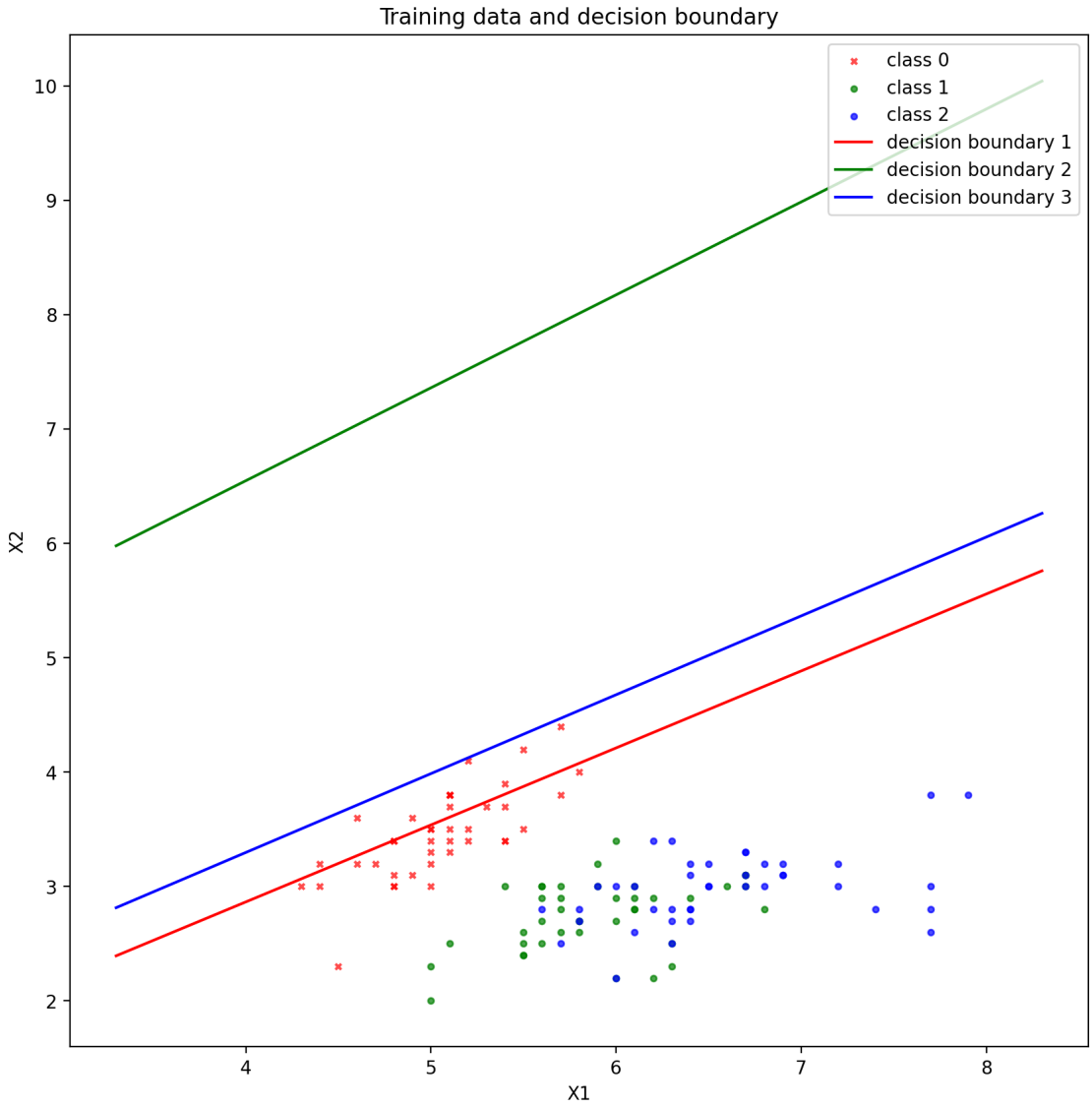
1.3.2 Results on Training set

[ ]:
```
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 %.
```

task3

April 20, 2023

1 Task 3

[ ]:
```
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 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 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
 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 Task3: Softmax on iris

Original target function:

$$L(\mathbf{w}_1, \dots, \mathbf{w}_K, b_1, \dots, b_K) = -\sum_i \ln p_{y^{(i)}} + \frac{\lambda}{2} \sum_{k=1}^{K} \|\mathbf{w}_k\|^2 .$$

$$\text{where } p_j = p(y = j|\mathbf{x}) = \frac{e^{f_j}}{\sum_{k=1}^{K} e^{f_k}}; f_j = \mathbf{w}_j \cdot \mathbf{x} + b_j$$

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

$$\frac{dL(\mathbf{w}_1, \dots, \mathbf{w}_K, b_1, \dots, b_K)}{d\mathbf{w}_k} = \lambda \mathbf{w}_k + \sum_{i, y_i = k} (p(k|\mathbf{x}_i) - 1)\mathbf{x}_i + \sum_{i, y_i \neq k} p(k|\mathbf{x}_i)\mathbf{x}_i.$$

$$\frac{dL(\mathbf{w}_1, \dots, \mathbf{w}_K, b_1, \dots, b_K)}{db_k} = \sum_{i, y_i = k} (p(k|\mathbf{x}_i) - 1) + \sum_{i, y_i \neq k} p(k|\mathbf{x}_i).$$

[ ]:
```
lamb=0.001 # Set the lambda for task3
learning_rate = 0.0001 # the alpha
n_iter = 20000
iterations = []
```

[ ]:
```
def softmax_P(X, W, b):
    f         = W.dot(X.T) + b    # Shape: [K,n].
    f         = f - f.max(axis=0, keepdims=True)  # Avoid the big number
    exp_f     = np.exp(f)         # Shape: [K,n].
    sum_exp_f = exp_f.sum(axis=0, keepdims=True)  # Shape: [1,n].
    P         = (exp_f / sum_exp_f).T
    return P
```

[ ]:
```
# gradient of loss function L(w)
# shape X (120, 5)
# Y (120,)
# W (3, 5)
# b (3, 1)
# P (120, 3)

def L_prime(X, Y, W, b):
    dL_by_dW = np.zeros_like(W)
    dL_by_db = np.zeros_like(b)

    P = softmax_P(X, W, b)
    for k in range(W.shape[0]):
        for i in range(X.shape[0]):
            if Y[i] == k:
```

```
                dL_by_dW[k,:] += (P[i,k] - 1) * X[i]
                dL_by_db[k,:] += (P[i,k] - 1)
            else:
                dL_by_dW[k,:] += (P[i,k]) * X[i]
                dL_by_db[k,:] += (P[i,k])
        dL_by_dW[k,:] += lamb * W[k]
    return dL_by_dW, dL_by_db
```

[ ]:
```
def L(X, Y, W, b):
    loss_sum = 0
    l2_sum = 0

    P = softmax_P(X, W, b)
    loss_sum = np.sum(np.log(P[range(Y.shape[0]), Y]))
    for k in range(W.shape[0]):
        l2_sum += np.dot(W[k].T, W[k])
    L = (lamb/2) * l2_sum - loss_sum
    return L
```

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

[ ]:
```
K = 3

# NOTE: Shape of w matrix is different from Task 2!
w = np.zeros((K, X_train.shape[1]))
b = np.zeros((K,1))    # Bias vector.

# We will keep track of training loss over iterations
iterations = [0]
L_list = [L(X_train, y_train, w, b)]
for i in range(n_iter):
    gradient_w, gradient_b = L_prime(X_train, y_train, w, b)
    w_new = w - learning_rate * gradient_w
    b_new = b - learning_rate * gradient_b
    iterations.append(i+1)
    L_list.append(L(X_train, y_train, w_new, b_new))

    if (np.linalg.norm(w_new - w, ord = 1) + np.linalg.norm(b_new - b, ord =␣
 ↪1)) < 0.0005:
        print("gradient descent has converged after " + str(i) + " iterations")
        break
    if i % 1000 == 0:
        print(i, np.linalg.norm(w_new - w, ord = 1) + np.linalg.norm(b_new - b,␣
 ↪ord = 1), L_list[-1])
    w = w_new
    b = b_new
```

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

```
0 0.018133333333333345 130.0969308576614
1000 0.0015265961323854133 40.64964008483113
2000 0.001045789989105983 30.64391021867177
3000 0.0008003824640225449 25.49768315357868
4000 0.0006503421506856566 22.330647367977292
5000 0.0005493829092290264 20.171031921771363
gradient descent has converged after 5648 iterations
w vector:
[[ 0.34088882  0.69679479  1.7436896  -2.4429943  -1.12047121]
 [ 0.5044756   0.53289235 -0.32021087  0.01421838 -0.99473911]
 [-0.84536442 -1.22968715 -1.42347872  2.42877592  2.11521032]]
b vector:
[[ 0.34102611]
 [ 0.50463694]
 [-0.84566305]]
```

1.3.2 Results on Training set

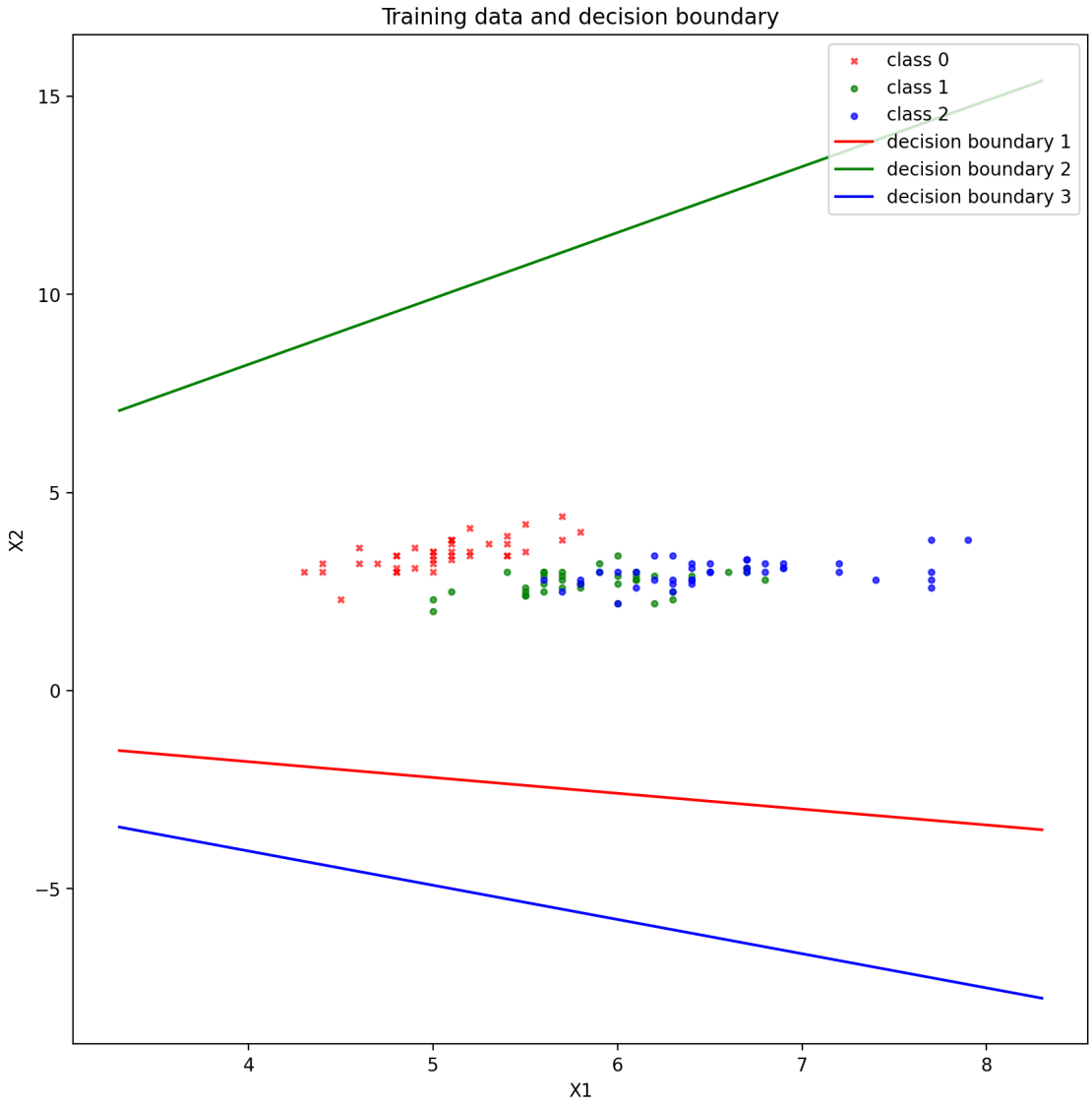
[ ]:
```
prediction = np.argmax(np.dot(X_train, w.T) + b.T,axis=1)
training_accuracy = (prediction - y_train == 0)

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 = (-b[0][0]-w[0][1]*x)/w[0][2]
y2 = (-b[1][0]-w[1][1]*x)/w[1][2]
y3 = (-b[2][0]-w[2][1]*x)/w[2][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 0x7fd1e80f2980>



1.3.3 Results on Test set

[ ]:
```
prediction = np.argmax(np.dot(X_test, w.T) + b.T, 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 %.
```