BÀI THỰC HÀNH PHẦN MULTI-CLASS SUPPORT VECTOR MACHINE

Ví dụ 1: Chúng ta tạo một bộ dữ liệu có 25000 mẫu (N=25000), mỗi mẫu có 3000 chiều (d=3000) được phân vào 10 lớp (C = 10), dữ liệu được tạo và phân lớp một cách ngẫu nhiên.

Tiếp theo chúng ta xây dựng các phương thức cần thiết cho phương pháp Multi-Class Vector Machine và tiến hành thực hiện phân lớp trên bộ dữ liệu nói trên. Trong chương trình này, chúng ta cài đặt phương pháp tiếp cận theo hướng Hingloss, tức là đưa về bài toán tối ưu toàn cục. Các bạn thực hành để kiểm nghiệm.

Trước hết ta gọi các thư viện cần thiết và tạo dữ liệu – Dữ liệu training sẽ là (X, y):

```
import numpy as np
from random import shuffle

N, C, d = 25000, 10, 3000
reg = .05
W = np.random.randn(d, C)
X = np.random.randn(d, N)
y = np.random.randint(C, size = N)
```

Tiếp theo chúng ta xây dựng phương thức tính hàm Loss (tổn thất) và đạo hàm (Grad) với dữ liệu hiện tại

```
# calculate loss and gradient base on original formula
# reg = 1/C
def svm_loss_grad (W, X, y, reg):
    d, C = W.shape
    _, N = X.shape

    ## Loss
    loss = 0
    ## Gradient
    dW = np.zeros_like(W)
    for n in xrange(N):
        xn = X[:, n]
        score = W.T.dot(xn)
        for j in xrange(C):
            if j == y[n]:
                continue
            # 1 - (b_yn + W_yn.x_n) + (b_j + W_j.x_n)
            margin = 1 - score[y[n]] + score[j]
            if margin > 0:
                loss += margin
                dW[:, j] += xn
                dW[:, y[n]] -= xn

    loss /= N
    loss += 0.5*reg*np.sum(W * W) # regularization + 1/2C * ||w||^2

    dW /= N
    dW += reg*W # gradient off regularization 1/C*w
    return loss, dW
```

Tiếp theo chúng ta sử dụng phương thức trên và quá trình lặp theo phương pháp Gradient Descent để tìm tham số W tối ưu (bài toán không ràng buộc):

```
# Mini-batch gradient descent
def multiclass_svm_GD(X, y, Winit, reg, lr=.1, \
        batch_size = 100, num_iters = 1000, print_every = 100):
    W = Winit
```

```
    loss_history = np.zeros((num_iters))
    for it in range(num_iters):
        # randomly pick a batch of X
        idx = np.random.choice(X.shape[1], batch_size)
        X_batch = X[:, idx]
        y_batch = y[idx]

        loss_history[it], dW = \
            svm_loss_grad(W, X_batch, y_batch, reg)

        W -= lr*dW
        if it % print_every == 1:
            print ('it %d/%d, loss = %f' %(it, num_iters, loss_history[it]))

    return W, loss_history
```

Cuối cùng, chúng ta chạy thử và in đồ thị hàm Loss để đánh giá kết quả:

```
W, loss_history = multiclass_svm_GD(X, y, W, reg)

import matplotlib.pyplot as plt
# plot loss as a function of iteration
plt.plot(loss_history)
plt.show()
```

Bài tập: Sử dụng các công cụ đánh giá cho bài toán phân loại để đánh giá kết quả.

Tính toán LOSS và GRADIENTS như phương thức phần trên là khá chậm vì chúng ta phải sử dụng vòng lặp tự xây dựng mà không vận dụng được các thư viện tính toán với ma trận/vector của NUMPY. Các thư viện này đều đã được xây dựng theo hướng tối ưu và song song hóa tính toán, nên sẽ hiệu quả hơn so với vòng lặp của chúng ta. Dưới đây ta xây dựng một phương thức tính toán theo hướng vector hóa để sử dụng các công cụ tính toán với ma trận/vector của NUMPY:

```
# method to compute loss function and its gradient
# here vectorization was used to speed up
def svm_loss_vectorized(W, X, y, reg):
    d, C = W.shape
    _, N = X.shape
    loss = 0
    dW = np.zeros_like(W)

    Z = W.T.dot(X)

    correct_class_score = np.choose(y, Z).reshape(N,1).T
    margins = np.maximum(0, Z - correct_class_score + 1)
    margins[y, np.arange(margins.shape[1])] = 0
# Loss function
    loss = np.sum(margins, axis = (0, 1))
    loss /= N
    loss += 0.5 * reg * np.sum(W * W)

    F = (margins > 0).astype(int)
    F[y, np.arange(F.shape[1])] = np.sum(-F, axis = 0)
# Gradient
    dW = X.dot(F.T)/N + reg*W
    return loss, dW
```

Xây dựng lại phương pháp lặp Mini Batch Gradient Descent sử dụng hàm tính Gradient và Loss đã vector hóa như trong phương thức trên

```
# Mini-batch gradient descent
def multiclass_svm_GD_vectorized (X, y, Winit, reg, lr=.1, \
        batch_size = 100, num_iters = 1000, print_every = 100):
    W = Winit
    loss_history = np.zeros((num_iters))
    for it in range(num_iters):
        # randomly pick a batch of X
        idx = np.random.choice(X.shape[1], batch_size)
        X_batch = X[:, idx]
        y_batch = y[idx]

        loss_history[it], dW = \
            svm_loss_vectorized(W, X_batch, y_batch, reg)

        W -= lr*dW
        if it % print_every == 1:
            print ('it %d/%d, loss = %f' %(it, num_iters, loss_history[it]))

    return W, loss_history
```

Các bạn sử dụng lại phần chạy thử và in ra biến thiên của Loss function, sau đó vẽ lên đồ thị để quan sát như ở phần trước, cần thay đổi tên phương thức phù hợp.

Bài tập thực hành 1:

Hãy tạo tập dữ liệu chỉ có 02 chiều, với các phân loại có phân bố chuẩn (xem lại ví dụ ở các phần trước), số mẫu mỗi loại là 200. Chạy lại chương trình trên với dữ liệu mới, sau đó hiển thị trực quan các điểm mẫu trên mặt phẳng để quan sát kết quả.

Bài tập thực hành 2:

Xem lại các bài ví dụ phần trước, sử dụng bộ trọng số W đã tính được và dự đoán các phân lớp y ứng với dữ liệu X trên chính tập dữ liệu ngẫu nhiên vừa tạo. Đưa ra độ chính xác Accuracy và ma trận nhầm lẫn Confusion Matrix.

Bài tập thực hành 3:

Hãy sửa phần đọc dữ liệu trong Ví dụ 1 ở trên, sau đó áp dụng cho phần phân loại chữ số viết tay với dữ liệu được cho trong bài tập phần Multinomial Logistic Regression (xem lại link sau đây để tìm hiểu cách đọc dữ liệu: https://classroom.google.com/u/1/c/MzkzNzg2MDI5MDcz/m/NDIxODAwNjQ5Njk5/details). Sau đó, hãy áp dụng phương pháp Multi-Class SVM với các đoạn lệnh nói trên để phân loại các chữ số viết tay đọc được.

Đưa ra độ chính xác, ma trận nhầm lẫn như với Bài tập thực hành 2.

Ví dụ 2. Chúng ta thực hiện với thư viện Sci-kit learn. Mã lệnh tương tự SoftMargin, chỉ số nhãn (labels) lớn hơn 2. Ta thực hiện ví dụ với dữ liệu nhân tạo dưới đây:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC

N = 1000  # Number of samples

# Create synthetic dataset
X1 = np.random.normal(loc=0, scale=1, size=(N, 2))
Y1 = 0 * np.ones(shape=(1000,))  # LABEL = 0
```

```
X2 = np.random.normal(loc=[-5, 5], scale=1, size=(N, 2))
Y2 = 1 * np.ones(shape=(1000,))  # LABEL = 1

X3 = np.random.normal(loc=[5, -5], scale=1, size=(N, 2))
Y3 = 2 * np.ones(shape=(1000,))  # LABEL = 2

# Create stacked dataset
X = np.vstack((X1, X2, X3))
Y = np.hstack((Y1, Y2, Y3))

# TRAIN SVM LEARNING ALGORITHM
clf = SVC(kernel='linear')
clf = clf.fit(X, Y)

# create decision boundary plot
xx, yy = np.meshgrid(
    np.arange(-10, 10, 0.2),
    np.arange(-10, 10, 0.2))
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# PLOT EVERYTHING
plt.scatter(X1[:,0], X1[:,1], color='r')
plt.scatter(X2[:,0], X2[:,1], color='b')
plt.scatter(X3[:,0], X3[:,1], color='y')
plt.contourf(xx,yy,Z,cmap=plt.cm.coolwarm, alpha=0.8)
plt.title("SVM for Three Labels (0, 1, 2)")
plt.show()
```

Bài tập tự thực hành 4: Hãy sử dụng đoạn code trên, tiến hành huấn luyện với 1000 ảnh từ tập ảnh chữ số viết tay và thực hiện dự đoán phân loại cho 500 ảnh khác.