```
import numpy as np
import pdb

"""
This code was based off of code from cs231n at Stanford University, and
    modified for ece239as at UCLA.
"""
class SVM(object):

  def __init__(self, dims=[10, 3073]):
    self.init_weights(dims=dims)

  def init_weights(self, dims):
    """
    Initializes the weight matrix of the SVM.  Note that it has shape (C, D)
    where C is the number of classes and D is the feature size.
    """
    self.W = np.random.normal(size=dims)

  def loss(self, X, y):
    """
    Calculates the SVM loss.

    Inputs have dimension D, there are C classes, and we operate on minibatches
    of N examples.

    Inputs:
    - X: A numpy array of shape (N, D) containing a minibatch of data.
    - y: A numpy array of shape (N,) containing training labels; y[i] = c means
      that X[i] has label c, where 0 <= c < C.

    Returns a tuple of:
    - loss as single float
    """

    # compute the loss and the gradient
    num_classes = self.W.shape[0]
    num_train = X.shape[0]
    loss = 0.0
    #print(self.W.shape)

    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the normalized SVM loss, and store it as 'loss'.
    #   (That is, calculate the sum of the losses of all the training
    #   set margins, and then normalize the loss by the number of
    #   training examples.)
    # ================================================================ #
    for i in np.arange(num_train):
        for j in np.arange(num_classes):
            if j == y[i]:
                continue
            aj = self.W[j].dot(X[i])
            ay = self.W[y[i]].dot(X[i])
            loss += max(0, 1 + aj - ay)
    loss = loss / num_train
```

```
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #

        return loss

    def loss_and_grad(self, X, y):
        """
        Same as self.loss(X, y), except that it also returns the gradient.

        Output: grad -- a matrix of the same dimensions as W containing
            the gradient of the loss with respect to W.
        """
        # compute the loss and the gradient
        num_classes = self.W.shape[0]
        num_train = X.shape[0]
        loss = 0.0
        delta = 1
        grad = np.zeros_like(self.W)

        # ================================================================ #
        # YOUR CODE HERE:
        #   Calculate the SVM loss and the gradient.  Store the gradient in
        #   the variable grad.ears
        # ================================================================ #
        for i in np.arange(num_train):
            for j in np.arange(num_classes):
                if j == y[i]:
                    continue
                aj = self.W[j].dot(X[i]) #incorrect class score
                ay = self.W[y[i]].dot(X[i]) #correct class score
                z = delta+aj-ay
                loss += max(0, z)
                if z > 0:
                    grad[j,:] +=X[i]
                    grad[y[i],:] -=X[i]

        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #
        #print (grad.shape) #(10,3073)
        loss /= num_train
        grad /= num_train

        return loss, grad

    def grad_check_sparse(self, X, y, your_grad, num_checks=10, h=1e-5):
        """
        sample a few random elements and only return numerical
        in these dimensions.
        """

        for i in np.arange(num_checks):
          ix = tuple([np.random.randint(m) for m in self.W.shape])

          oldval = self.W[ix]
```

```
        self.W[ix] = oldval + h # increment by h
        fxph = self.loss(X, y)
        self.W[ix] = oldval - h # decrement by h
        fxmh = self.loss(X,y) # evaluate f(x - h)
        self.W[ix] = oldval # reset

        grad_numerical = (fxph - fxmh) / (2 * h)
        grad_analytic = your_grad[ix]
        rel_error = abs(grad_numerical - grad_analytic) / (abs(grad_numerical) +
            abs(grad_analytic))
        print('numerical: %f analytic: %f, relative error: %e' % (grad_numerical,
            grad_analytic, rel_error))

    def fast_loss_and_grad(self, X, y):
      """
      A vectorized implementation of loss_and_grad. It shares the same
      inputs and ouptuts as loss_and_grad.
      """
      num_classes = self.W.shape[0]
      num_train = X.shape[0]
      delta = 1
      loss = 0.0
      grad = np.zeros(self.W.shape) # initialize the gradient as zero

      # ================================================================ #
      # YOUR CODE HERE:
      #   Calculate the SVM loss WITHOUT any for loops.
      # ================================================================ #
      scores = np.matmul(self.W,(X.T))
      #print('score shape', scores.shape) #(10,500)
      correct_scores = scores[y,np.arange(num_train)]
      #print('correct score shape', correct_scores.shape) #(500,)
      z = delta+scores-correct_scores
      margins = np.maximum(0, z) #broadcasting
      #print(margins.shape) (10,500)
      margins[y, np.arange(num_train)] = 0 #correct class score rows should be 0
      loss = np.sum(margins)
      loss = loss / num_train
      # ================================================================ #
      # END YOUR CODE HERE
      # ================================================================ #
      # ================================================================ #
      # YOUR CODE HERE:
      #   Calculate the SVM grad WITHOUT any for loops.
      # ================================================================ #
      grad_scores = np.zeros(margins.shape)
      grad_scores[margins > 0] = 1 #indicator function gives 1 if z>0, gives 0 if
          <=0
      #print(grad_scores.shape) #(10,500)
      grad_scores[y, np.arange(num_train)] = -np.sum(margins > 0, axis = 0)
      #print (X.shape) (500,3073)
      grad = np.matmul(grad_scores,(X))
      grad = grad / num_train
      # ================================================================ #
      # END YOUR CODE HERE
      # ================================================================ #
```

```
    return loss, grad

  def train(self, X, y, learning_rate=1e-3, num_iters=100,
            batch_size=200, verbose=False):
    """
    Train this linear classifier using stochastic gradient descent.

    Inputs:
    - X: A numpy array of shape (N, D) containing training data; there are N
      training samples each of dimension D.
    - y: A numpy array of shape (N,) containing training labels; y[i] = c
      means that X[i] has label 0 <= c < C for C classes.
    - learning_rate: (float) learning rate for optimization.
    - num_iters: (integer) number of steps to take when optimizing
    - batch_size: (integer) number of training examples to use at each step.
    - verbose: (boolean) If true, print progress during optimization.

    Outputs:
    A list containing the value of the loss function at each training iteration
        .
    """
    num_train, dim = X.shape
    num_classes = np.max(y) + 1 # assume y takes values 0...K-1 where K is
        number of classes

    self.init_weights(dims=[np.max(y) + 1, X.shape[1]]) # initializes the
        weights of self.W

    # Run stochastic gradient descent to optimize W
    loss_history = []

    for it in np.arange(num_iters):
      X_batch = None
      y_batch = None

      # ================================================================ #
      # YOUR CODE HERE:
      #   Sample batch_size elements from the training data for use in
      #   gradient descent.  After sampling,
      #     - X_batch should have shape: (batch_size, dim)
      #     - y_batch should have shape: (batch_size,)
      #   The indices should be randomly generated to reduce correlations
      #   in the dataset.  Use np.random.choice.  It's okay to sample with
      #   replacement.
      # ================================================================ #
      indices = np.random.choice(np.arange(num_train), batch_size)
      #print('indices shape', indices.shape) #(200,)
      X_batch = X[indices,:]
      #print('X_batch shape', X_batch.shape) #(200,3073)
      y_batch = y[indices]
      #print('y_batch shape', y_batch.shape) #(200,)

      # ================================================================ #
      # END YOUR CODE HERE
      # ================================================================ #
```

```
    # evaluate loss and gradient
    loss, grad = self.fast_loss_and_grad(X_batch, y_batch)
    loss_history.append(loss)

    # ================================================================== #
    # YOUR CODE HERE:
    #   Update the parameters, self.W, with a gradient step
    # ================================================================== #
    self.W -=learning_rate * grad
    # ================================================================== #
    # END YOUR CODE HERE
    # ================================================================== #

    if verbose and it % 100 == 0:
      print('iteration {} / {}: loss {}'.format(it, num_iters, loss))

  return loss_history

def predict(self, X):
  """
  Inputs:
  - X: N x D array of training data. Each row is a D-dimensional point.

  Returns:
  - y_pred: Predicted labels for the data in X. y_pred is a 1-dimensional
    array of length N, and each element is an integer giving the predicted
    class.
  """
  y_pred = np.zeros(X.shape[1])


  # ================================================================== #
  # YOUR CODE HERE:
  #   Predict the labels given the training data with the parameter self.W.
  # ================================================================== #
  scores = X.dot(self.W.T)
  #print(scores.shape)
  y_pred = np.argmax(scores, axis = 1)
  # ================================================================== #
  # END YOUR CODE HERE
  # ================================================================== #

  return y_pred
```