```
import numpy as np
import scipy.sparse
import scipy.optimize
import utils
import this

class SoftmaxRegression:
    """
    Here you will fill in this incomplete implementation of Softmax regression.

    Adapted from code by Jatin Shah
    """
    def __init__(self, numClasses, exSize, opts={'maxIter':400}):
        """
        numClasses:     number of possible classifications
        exSize:         size of attribute array (number of input features)
        reg:            regularizing term coefficient (lambda)
        opts:           in this class the only option used is maxIter
        """

        self.numClasses = numClasses
        self.exSize = exSize
        self.opts = opts

        # Initialize weight matrix with empty matrix
        self.W = np.zeros((numClasses, exSize))

        #self.W = 0.005 * np.random.randn(numClasses, exSize)

    def reset(self, numClasses, exSize, opts={'maxIter':400}):
        self.__init__(numClasses, exSize, opts)

    def setOption(self, optName, optVal):
        """
        optName:        name of option
        optVal:         new value to assign option to
        """

        self.opts[optName] = optVal

    def cost(self, X, Y, W=None):
        """
        Calculate the cost function for X and Y using current weight matrix W. Note
that we are not using
        a regularizer in the cost; this is equivalent to lambda = 0.

        X:              (M x N) matrix of input feature values,
                            where M = exSize, N = number of examples
        Y:              (N x 1) array of expected output classes for each example

        Returns the cost and its gradient, which is the form needed to use
scipy.optimize.minimize
        """

        if W is None:
            W = this.W
        numClasses = self.numClasses
        exSize = self.exSize
```

```
        W = W.reshape(numClasses, exSize)           # Ensure W is in the correct
dimensions
        N = X.shape[1]                              # N = number of examples

        W_X = W.dot(X)                              # This is our activation matrix
with dimensions (A * B)
                                                    # where A is the number of
classes and B is the number
                                                    # of examples. (W_X[a, b] gives
the activation of example
                                                    # b for class a.) You will use
this matrix to find the
                                                    # probabilities that example b
is class a using the
                                                    # softmax formula.

        W_X = W_X - np.max(W_X)

        # This is the indicator function used in the loss function, where
indicator[a, b] = 1
        # when example b is labeled a (according to the target Y) and indicator[a,
b] = 0 otherwise.

        indicator = scipy.sparse.csr_matrix((np.ones(N), (Y, np.array(range(N)))))
        indicator = np.resize(np.array(indicator.todense()), (numClasses, N))

        # TODO: Compute the predicted probabilities, the total cost, and the
gradient.

        # Each column of W_X is the set of activations for each class corresponding
to
        # one example; the probabilties are given by the exponential of each entry
        # divided by the sum of the exponentials over the entire column.

        # The cost associated with a single example is given by -1 times the log
probability
        # of the true class; initialize the cost variable to the AVERAGE cost over
all the examples.
        # Hint: there's an easy way to do this with the indicator matrix.

        # The gradient has the same dimensions as W, and each component (i,j)
represents the
        # derivative of the cost with respect to the weight associated with class
i, attribute j.
        # The gradient associated with a single example x is given by -1 * A * x_T,
where x_T is
        # the transpose of the example, and A is a vector with component i given by
(1 - P(class = i))
        # if the true class is i, and (-P(class = i)) otherwise. Notice that this
multiplication gives
        # the desired dimensions. Find the AVERAGE gradient over all the examples.
Again, there is
        # an easy way to do this with the indicator matrix.

        ### YOUR CODE HERE ###

        ex = np.exp(W_X)
        probabilities = ex/(np.sum(ex, axis=0))
        cost = (-1 * np.multiply(indicator, np.log(probabilities)).sum())/N
```

```
        gradient = (-1 * ((indicator - probabilities).dot(X.T)))/N

        ### YOUR CODE HERE ###

        # flatten is needed by scipy.optimize.minimize
        return cost, gradient.flatten()


    def train(self, X, Y):
        """
        Train to find optimal weight matrix W. Here we make use of the SciPy
optimization library but
        in theory you could implement gradient descent to do this as well.

        X:              (M x N) matrix of input feature values,
                            where M = exSize, N = number of examples
        Y:              (N x 1) array of expected output classes for each example
        maxIter:        Maximum training iterations
        """

        numClasses = self.numClasses
        exSize = self.exSize
        W = self.W

        # Set maxIter hyperparameter
        if self.opts['maxIter'] is None:
            self.opts['maxIter'] = 400

        # Lambda function needed by scipy.optimize.minimize
        J = lambda w: self.cost(X, Y, w)

        # SciPy is a powerful data science library, check it out if you're
interested :)
        result = scipy.optimize.minimize(J, W, method='L-BFGS-B', jac=True,
options={'maxiter': self.opts['maxIter'], 'disp': True})
        self.W = result.x       # save the optimal solution found

    def predict(self, X):
        """
        Use W to predict the classes of each example in X.

        X:              (M x N) matrix of input feature values,
                            where M = exSize, N = number of examples

        """

        W = self.W.reshape(self.numClasses, self.exSize)
        W_X = W.dot(X)

        # TODO: Compute the predicted probabilities and the predicted classes for
each example
        # Reminder: The predicted class for a single example is just the one with
the highest probability

        ### YOUR CODE HERE ###

        ex = np.exp(W_X)
        probabilities = ex/(np.sum(ex, axis=0))
        predicted_classes = np.argmax(probabilities, axis=0)
```

```
### YOUR CODE (ENDS) HERE ###

return predicted_classes
```