```python
"""
Author      : Madison Hobbs and Shota Yasunaga
Class       : HMC CS 158
Date        : 2018 Mar 05
Description : Multiclass Classification on Soybean Dataset
              This code was adapted from course material by Tommi Jaakola (MIT)
"""

# utilities
from util import *

# scikit-learn libraries
from sklearn.svm import SVC
from sklearn import metrics

######################################################################
# output code functions
######################################################################

def generate_output_codes(num_classes, code_type) :
    """
    Generate output codes for multiclass classification.

    For one-versus-all
        num_classifiers = num_classes
        Each binary task sets one class to +1 and the rest to -1.
        R is ordered so that the positive class is along the diagonal.

    For one-versus-one
        num_classifiers = num_classes choose 2
        Each binary task sets one class to +1, another class to -1, and the rest to
0.
        R is ordered so that
          the first class is positive and each following class is successively negat
ive
          the second class is positive and each following class is successively nega
tie
          etc

    Parameters
    --------------------
        num_classes     -- int, number of classes
        code_type       -- string, type of output code
                           allowable: 'ova', 'ovo'

    Returns
    --------------------
        R               -- numpy array of shape (num_classes, num_classifiers),
                           output code
    """

    ### ========== TODO : START ========== ###
    # part a: generate output codes
    # hint : initialize with np.ones(...) and np.zeros(...)

    if code_type == "ova":
        # basically an identity matrix but the zeros are -1's
        # each class gets to be the star (value 1) only once.
        R = -np.ones((num_classes, num_classes)) + 2*np.identity(num_classes)

    elif code_type == "ovo" :
        # each classifier only deals with two classes
        n_choose_2 = int((num_classes-1)*num_classes/float(2))
        R = np.zeros((num_classes, n_choose_2))

        column = 0
        for posRow in range(num_classes) :
            negRow = posRow + 1
            while negRow < num_classes :
                R[posRow, column] = 1
                R[negRow, column] = -1
                negRow += 1
                column += 1
```

```
    else: R = None
    ### ========== TODO : END ========== ###

    return R


def load_code(filename) :
    """
    Load code from file.

    Parameters
    --------------------
        filename -- string, filename
    """

    # determine filename
    import util
    dir = os.path.dirname(util.__file__)
    f = os.path.join(dir, '..', 'data', filename)

    # load data
    with open(f, 'r') as fid :
        data = np.loadtxt(fid, delimiter=",")

    return data


def test_output_codes():
    R_act = generate_output_codes(3, 'ova')
    R_exp = np.array([[  1, -1, -1],
                      [ -1,  1, -1],
                      [ -1, -1,  1]])
    assert (R_exp == R_act).all(), "'ova' incorrect"

    R_act = generate_output_codes(3, 'ovo')
    R_exp = np.array([[  1,  1,  0],
                      [ -1,  0,  1],
                      [  0, -1, -1]])
    assert (R_exp == R_act).all(), "'ovo' incorrect"


######################################################################
# loss functions
######################################################################

def compute_losses(loss_type, R, discrim_func, alpha=2) :
    """
    Given output code and distances (for one example), compute losses (for each clas
s).

    hamming  : Loss  = (1 - sign(z)) / 2
    sigmoid  : Loss = 1 / (1 + exp(alpha * z))
    logistic : Loss = log(1 + exp(-alpha * z))

    Parameters
    --------------------
        loss_type    -- string, loss function
                        allowable: 'hamming', 'sigmoid', 'logistic'
        R            -- numpy array of shape (num_classes, num_classifiers)
                        output code
        discrim_func -- numpy array of shape (num_classifiers,)
                        distance of sample to hyperplanes, one per classifier
        alpha        -- float, parameter for sigmoid and logistic functions

    Returns
    --------------------
        losses       -- numpy array of shape (num_classes,), losses
    """

    # element-wise multiplication of matrices of shape (num_classes, num_classifiers
)
```

```
    # tiled matrix created from (vertically) repeating discrim_func num_classes time
s
    z = R * np.tile(discrim_func, (R.shape[0],1))    # element-wise

    # compute losses in matrix form
    if loss_type == 'hamming' :
        losses = np.abs(1 - np.sign(z)) * 0.5

    elif loss_type == 'sigmoid' :
        losses = 1./(1 + np.exp(alpha * z))

    elif loss_type == 'logistic' :
        # compute in this way to avoid numerical issues
        # log(1 + exp(-alpha * z)) = -log(1 / (1 + exp(-alpha * z)))
        eps = np.spacing(1) # numpy spacing(1) = matlab eps
        val = 1./(1 + np.exp(-alpha * z))
        losses = -np.log(val + eps)

    else :
        raise Exception("Error! Unknown loss function!")

    # sum over losses of binary classifiers to determine loss for each class
    losses = np.sum(losses, 1) # sum over each row

    return losses


def hamming_losses(R, discrim_func) :
    """
    Wrapper around compute_losses for hamming loss function.
    """
    return compute_losses('hamming', R, discrim_func)


def sigmoid_losses(R, discrim_func, alpha=2) :
    """
    Wrapper around compute_losses for sigmoid loss function.
    """
    return compute_losses('sigmoid', R, discrim_func, alpha)


def logistic_losses(R, discrim_func, alpha=2) :
    """
    Wrapper around compute_losses for logistic loss function.
    """
    return compute_losses('logistic', R, discrim_func, alpha)


######################################################################
# classes
######################################################################

class MulticlassSVM :

    def __init__(self, R, C=1.0, kernel='linear', **kwargs) :
        """
        Multiclass SVM.

        Attributes
        --------------------
            R       -- numpy array of shape (num_classes, num_classifiers)
                       output code
            svms    -- list of length num_classifiers
                       binary classifiers, one for each column of R
            classes -- numpy array of shape (num_classes,) classes

        Parameters
        --------------------
            R       -- numpy array of shape (num_classes, num_classifiers)
                       output code
            C       -- numpy array of shape (num_classifiers,1) or float
                       penalty parameter C of the error term
            kernel  -- string, kernel type
```

```
                      see SVC documentation
            kwargs   -- additional named arguments to SVC
        """

        num_classes, num_classifiers = R.shape

        # store output code
        self.R = R

        # use first value of C if dimension mismatch
        try :
            if len(C) != num_classifiers :
                raise Warning("dimension mismatch between R and C " +
                                "==> using first value in C")
                C = np.ones((num_classifiers,)) * C[0]
        except :
            C = np.ones((num_classifiers,)) * C

        # set up and store classifier corresponding to jth column of R
        self.svms = [None for _ in xrange(num_classifiers)]
        for j in xrange(num_classifiers) :
            svm = SVC(kernel=kernel, C=C[j], **kwargs)
            self.svms[j] = svm


    def fit(self, X, y) :
        """
        Learn the multiclass classifier (based on SVMs).

        Parameters
        --------------------
            X    -- numpy array of shape (n,d), features
            y    -- numpy array of shape (n,), targets

        Returns
        --------------------
            self -- an instance of self
        """

        classes = np.unique(y)
        num_classes, num_classifiers = self.R.shape
        if len(classes) != num_classes :
            raise Exception('num_classes mismatched between R and data')
        self.classes = classes    # keep track for prediction

        ### ========== TODO : START ========== ###
        # part c: train binary classifiers

        # HERE IS ONE WAY (THERE MAY BE OTHER APPROACHES)
        for classifier in range(num_classifiers):
            # keep two lists, pos_ndx and neg_ndx, that store indices
        #   of examples to classify as pos / neg for current binary task
            pos_ndx = np.zeros(0,dtype=int) #store row
            neg_ndx = np.zeros(0,dtype= int)

            # for each class C
            for i in range(len(self.classes)) :
        # a) find indices for which examples have class equal to C
        #    [use np.nonzero(CONDITION)[0]]p

        # b) update pos_ndx and neg_ndx based on output code R[i,j]
        #    where i = class index, j = classifier index
                if self.R[i, classifier] == 1 :
                    rows = np.nonzero(y == self.classes[i])[0]
                    pos_ndx = np.append(pos_ndx, rows)
                if self.R[i, classifier] == -1 :
                    rows = np.nonzero(y == self.classes[i])[0]
                    neg_ndx = np.append(neg_ndx, rows)
        # set X_train using X with pos_ndx and neg_ndx
            train_index = np.append(pos_ndx,neg_ndx)
            X_train = (X[train_index,:])
        # set y_train using y with pos_ndx and neg_ndx
            y_train = np.append(np.ones(len(pos_ndx)), -np.ones(len(neg_ndx)))
```

```
        #     y_train should contain only {+1,-1}
        #
        # train the binary classifier
            self.svms[classifier].fit(X_train, y_train)


        return self

        ### ========== TODO : END ========== ###


    def predict(self, X, loss_func=hamming_losses) :
        """
        Predict the optimal class.

        Parameters
        --------------------
            X         -- numpy array of shape (n,d), features
            loss_func -- loss function
                         allowable: hamming_losses, logistic_losses, sigmoid_losses

        Returns
        --------------------
            y         -- numpy array of shape (n,), predictions
        """

        n,d = X.shape
        num_classes, num_classifiers = self.R.shape

        # setup predictions
        y = np.zeros(n)

        ### ========== TODO : START ========== ###
        # part d: predict multiclass class
        #
        # HERE IS ONE WAY (THERE MAY BE OTHER APPROACHES)
        #
        # for each example
        discrim_func = np.zeros(n)
        #   predict distances to hyperplanes using SVC.decision_function(...)
        for svm in self.svms :
            distance = svm.decision_function(X)
            discrim_func = np.vstack((discrim_func, distance))
        #   find class with minimum loss (be sure to look up in self.classes)
        #   if you have a choice between multiple occurrences of the minimum values,
        #   use the index corresponding to the first occurrence

        for row in range(n):
            losses = loss_func(self.R, discrim_func[1:,row])
            pred_class_ndx = np.argmin(losses) # index of minimum loss
            y[row] = self.classes[pred_class_ndx]

        ### ========== TODO : END ========== ###

        return y

######################################################################
# main
######################################################################

def main() :
    # load data
    converters = {35: ord} # label (column 35) is a character
    train_data = load_data("soybean_train.csv", converters)
    test_data = load_data("soybean_test.csv", converters)
    num_classes = 15


    ### ========== TODO : START ========== ###

    # part b : generate output codes
    test_output_codes()
```

```
    # graph
    z = np.arange(-2, 2, 0.1)

    # exponential : Loss = exp(z)
    #expo = np.exp(-z)
    #plt.plot(z, expo, label = "exponential loss")
    #plt.legend()
    #plt.show()

    # hamming  : Loss  = (1 - sign(z)) / 2
    hamming = np.abs(1 - np.sign(z)) * 0.5
    # sigmoid  : Loss = 1 / (1 + exp(alpha * z))
    sig1 = 1./(1 + np.exp(1 * z))
    sig2 = 1./(1 + np.exp(2 * z))
    # logistic : Loss = log(1 + exp(-alpha * z))
    alpha = 1
    eps = np.spacing(1) # numpy spacing(1) = matlab eps
    val = 1./(1 + np.exp(-1 * z))
    log1 = -np.log(val + eps)

    alpha = 2
    eps = np.spacing(1) # numpy spacing(1) = matlab eps
    val = 1./(1 + np.exp(-2 * z))
    log2 = -np.log(val + eps)

    plt.plot(z, hamming, label = "hamming")
    plt.plot(z, sig1, label = "sigmoid, alpha = 1")
    plt.plot(z, sig2, label = "sigmoid, alpha = 2")
    plt.plot(z, log1, label = "logistic, alpha = 1")
    plt.plot(z, log2, label = "logistic, alpha = 2")
    plt.legend()
    plt.show()

    # parts c-e : train component classifiers, make predictions,
    #             compare output codes and loss functions
    #
    # use generate_output_codes(...) to generate OVA and OVO codes
    loss_func_list = [hamming_losses,sigmoid_losses,logistic_losses]
    R_list = [generate_output_codes(num_classes, 'ova'),generate_output_codes(num_cl
asses, 'ovo'),load_code("R1.csv"), load_code("R2.csv")]
    code_itr = iter(['ova','ovo','R1','R2']*3)
    # use load_code(...) to load random codes
    #
    import warnings
    warnings.filterwarnings("ignore",category=DeprecationWarning)
    # for each output code and loss function

    for loss_func in loss_func_list :
        for R in R_list :
    #   train a multiclass SVM on training data and evaluate on test data
    #   setup the binary classifiers using the specified parameters from the handout
            clf = MulticlassSVM(R = R, kernel='poly', degree = 4, coef0 = 1, gamma =
 1.0)
            clf.fit(train_data.X, train_data.y)
            pred = clf.predict(test_data.X, loss_func=loss_func)
            print '-----------------'
            print '                 '
            print str(loss_func)
            print code_itr.next()
            print "support vecs:1  " + str(clf.svms[0].support_)
            print "support vecs:2  " + str(clf.svms[1].support_)
            num_errors = sum(pred != test_data.y)
            print "number of erros: " + str(num_errors)
    # if you implemented MulticlassSVM.fit(...) correctly,
    #   using OVA, your first trained binary classifier
    #   should have the following indices for support vectors
    #     array([ 12,  22,  29,  37,  41,  44,  49,  55,  76, 134,
    #            157, 161, 167, 168,   0,   3,   7])
    #
    # if you implemented MulticlassSVM.predict(...) correctly,
    #   using OVA and Hamming loss, you should find 54 errors
```

```
    ### ========== TODO : END ========== ###

if __name__ == "__main__" :
    main()
```