03/23/18 02:15:59
/Users/caiglencross/Documents/MachineLearning/ps2/ps7/source/soybean.py

```
"""
Author      : Cai Glencross & Katie Li
Class       : HMC CS 158
Date        : 2018 Mar 23
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
import math


######################################################################
##
# 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 negative
          the second class is positive and each following class is
successively negatie
          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'):
        R = -1* np.ones((num_classes, num_classes)) + 2*np.identity(num_classes)
    if (code_type == 'ovo'):
        #generate the correct number of rows
        f = math.factorial
        n = num_classes
        #find n choose 2 classes
        nC2_classes = f(n) / f(2) /f(n-2)
        R = np.zeros((num_classes, nC2_classes))
        current_col = 0
        for i in range(num_classes):
            for j in range(i+1,num_classes):
                R[i, current_col] = 1
                R[j, current_col] = -1
                current_col = current_col + 1
    ### ========== 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
```

```python
    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 class).

    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 times
    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)
        #
        # keep two lists, pos_ndx and neg_ndx, that store indices
        #   of examples to classify as pos / neg for current binary task
        #
        # for each class C
        # a) find indices for which examples have class equal to C
        #    [use np.nonzero(CONDITION)[0]]
        # b) update pos_ndx and neg_ndx based on output code R[i,j]
        #    where i = class index, j = classifier index
        #
        # set X_train using X with pos_ndx and neg_ndx
        # set y_train using y with pos_ndx and neg_ndx
        #     y_train should contain only {+1,-1}
```

```
        #
        # train the binary classifier
        n,d = X.shape
        R = self.R

        for i in range(0, len(self.svms)):
            pos_ndx = []
            neg_ndx = []
            for ndx in range(0, n):
                class_index = list(classes).index(y[ndx])
                if R[class_index, i] == 1:
                    pos_ndx.append(ndx)
                elif R[class_index, i] == -1:
                    neg_ndx.append(ndx)

            new_X_pos = X[pos_ndx,:]
            new_X_neg = X[neg_ndx,:]

            #get the new_X with labels
            new_X = np.vstack((new_X_pos, new_X_neg))
            new_Y = np.append(
                    np.ones(len(pos_ndx)), (-1*np.ones(len(neg_ndx))) )

            self.svms[i].fit(new_X, new_Y)
        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
        #   predict distances to hyperplanes using SVC.decision_function(...)
        #   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 i in range(n):
            dists = np.empty(num_classifiers)
            for j in range(num_classifiers):
                X_i_reshaped = X[i,:].reshape(1,-1)
                dist = self.svms[j].decision_function(X_i_reshaped)

                dists[j] = dist


            losses = loss_func(self.R, dists)
            best_class_index = np.argmin(losses)
            y[i] = self.classes[best_class_index]

        ### ========== 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

    # part b : generate output codes
    test_output_codes()

    # plot loss functions
```

```
    z = np.arange(-2, 3, 0.01)
    hamming = map(lambda u: (1 - np.sign(u))/2, z)
    sigmoid1 = map(lambda u: (1/ (1+np.exp(u))), z)
    sigmoid2 = map(lambda u: (1/ (1+np.exp(2*u))), z)
    logistic1 = map(lambda u: (math.log(1 + np.exp(-u))), z)
    logistic2 = map(lambda u: (math.log(1 + np.exp(-2*u))), z)

    plt.plot(z, hamming, label="Hamming")
    plt.plot(z, sigmoid1, label="Sigmoid1")
    plt.plot(z, sigmoid2, label="Sigmoid2")
    plt.plot(z, logistic1, label="Logistic1")
    plt.plot(z, logistic2, label="Logistic2")
    plt.legend()
    #plt.show()


    ### ========== TODO : START ========== ###
    # parts c-e : train component classifiers, make predictions,
    #             compare output codes and loss functions
    #
    # use generate_output_codes(...) to generate OVA and OVO codes
    # use load_code(...) to load random codes
    #
    # for each output code and loss function
    #   train a multiclass SVM on training data and evaluate on test data
    #   setup the binary classifiers using the specified parameters from the handout
    #
    # 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

    ova = generate_output_codes(len(np.unique(train_data.y)), 'ova')

    multiclass = MulticlassSVM(ova, C=10, kernel='poly', gamma=1, degree=4, coef0=1)
    multiclass = multiclass.fit(train_data.X, train_data.y)

    predictions = multiclass.predict(test_data.X, loss_func=hamming_losses)

    errors = metrics.zero_one_loss(predictions, test_data.y, normalize=False)
    print(errors)
```

```python

    print(multiclass.svms[0].support_)
    ### ========== TODO : END ========== ###

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