```
"""
Author      : Savannah Baron and Varsha Kishore
Class       : HMC CS 158
Date        : 2017 March 7
Description : Project 6
"""

"""
Author      : Yi-Chieh Wu
Class       : HMC CS 158
Date        : 2017 Feb 13
Description : Twitter
"""

from string import punctuation
from matplotlib.pylab import *

import numpy as np

from sklearn.svm import SVC
from sklearn.cross_validation import StratifiedKFold
from sklearn import metrics
from sklearn.utils import shuffle
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer


######################################################################
# functions -- input/output
######################################################################

def read_vector_file(fname):
    """
    Reads and returns a vector from a file.

    Parameters
    --------------------
        fname  -- string, filename

    Returns
    --------------------
        labels -- numpy array of shape (n,)
                    n is the number of non-blank lines in the text file
    """
    return np.genfromtxt(fname)


def write_label_answer(vec, outfile):
    """
```

```
    Writes your label vector to the given file.

    Parameters
    --------------------
        vec     -- numpy array of shape (n,) or (n,1), predicted scores
        outfile -- string, output filename
    """

    # for this project, you should predict 70 labels
    if(vec.shape[0] != 70):
        print("Error - output vector should have 70 rows.")
        print("Aborting write.")
        return

    np.savetxt(outfile, vec)


######################################################################
# functions -- feature extraction
######################################################################

def extract_words(input_string):
    """
    Processes the input_string, separating it into "words" based on the presence
    of spaces, and separating punctuation marks into their own words.

    Parameters
    --------------------
        input_string -- string of characters

    Returns
    --------------------
        words        -- list of lowercase "words"
    """

    for c in punctuation :
        input_string = input_string.replace(c, ' ' + c + ' ')
    return input_string.lower().split()


def extract_dictionary(infile):
    """
    Given a filename, reads the text file and builds a dictionary of unique
    words/punctuations.

    Parameters
    --------------------
        infile    -- string, filename
```

```
    Returns
    --------------------
        word_list -- dictionary, (key, value) pairs are (word, index)
    """
    word_list = {}
    with open(infile, 'rU') as fid :
        ### ========== TODO : START ========== ###
        # part 1a: process each line to populate word_list
        counter = 0
        for  line in fid:
        words = extract_words(line)
        for word in words:
         if not word in word_list:
         word_list[word] = counter
         counter +=1
        ### ========== TODO : END ========== ###

    return word_list


def extract_feature_vectors(infile, word_list):
    """
    Produces a bag-of-words representation of a text file specified by the
    filename infile based on the dictionary word_list.

    Parameters
    --------------------
        infile         -- string, filename
        word_list      -- dictionary, (key, value) pairs are (word, index)

    Returns
    --------------------
        feature_matrix -- numpy array of shape (n,d)
                          boolean (0,1) array indicating word presence in a string
                            n is the number of non-blank lines in the text file
                            d is the number of unique words in the text file
    """

    num_lines = sum(1 for line in open(infile,'rU'))
    num_words = len(word_list)
    feature_matrix = np.zeros((num_lines, num_words))

    with open(infile, 'rU') as fid :
        ### ========== TODO : START ========== ###
        # part 1b: process each line to populate feature_matrix
        counter = 0
        for line in fid:
```

```
        words = extract_words(line)
        for word in words:
        if word in word_list:
        feature_matrix[counter, word_list[word]] = 1
        counter += 1

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

    return feature_matrix


def test_extract_dictionary(dictionary) :
    err = "extract_dictionary implementation incorrect"

    assert len(dictionary) == 1811, err

    exp = [('2012', 0),
           ('carol', 10),
           ('ve', 20),
           ('scary', 30),
           ('vacation', 40),
           ('just', 50),
           ('excited', 60),
           ('no', 70),
           ('cinema', 80),
           ('frm', 90)]
    act = [sorted(dictionary.items(), key=lambda it: it[1])[i] for i in range(0,100,10)]
    assert exp == act, err


def test_extract_feature_vectors(X) :
    err = "extract_features_vectors implementation incorrect"

    assert X.shape == (630, 1811), err

    exp = np.array([[ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
                    [ 1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.],
                    [ 0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  1.],
                    [ 0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,  1.,  1.],
                    [ 0.,  1.,  0.,  0.,  0.,  1.,  0.,  0.,  1.,  1.],
                    [ 0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  1.,  1.],
                    [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  1.],
                    [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  1.],
                    [ 0.,  1.,  0.,  0.,  1.,  0.,  0.,  0.,  1.,  1.],
                    [ 0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.]])
    act = X[:10,:10]
    assert (exp == act).all(), err
```

```
######################################################################
# functions -- evaluation
######################################################################

def performance(y_true, y_pred, metric="accuracy"):
    """
    Calculates the performance metric based on the agreement between the
    true labels and the predicted labels.

    Parameters
    --------------------
        y_true -- numpy array of shape (n,), known labels
        y_pred -- numpy array of shape (n,), (continuous-valued) predictions
        metric -- string, option used to select the performance measure
                  options: 'accuracy', 'f1-score', 'auroc', 'precision',
                           'sensitivity', 'specificity'

    Returns
    --------------------
        score  -- float, performance score
    """
    # map continuous-valued predictions to binary labels
    y_label = np.sign(y_pred)
    y_label[y_label==0] = 1 # map points of hyperplane to +1

    ### ========== TODO : START ========== ###
    # part 2a: compute classifier performance
    #C_00 = TN, C_10 = FN, C_11 = TP, C_01 = FP
    cm = metrics.confusion_matrix(y_true, y_label)
    if metric == "accuracy":
     return metrics.accuracy_score(y_true, y_label)

    elif metric == "f1_score":
     return metrics.f1_score(y_true, y_label)

    elif metric == "auroc":
     return metrics.roc_auc_score(y_true, y_pred)

    elif metric == "precision":
     return cm[1,1] / float(cm[1,1]+ cm[0,1])

    elif metric == "sensitivity":
     return cm[1,1] / float(cm[1,1] + cm[1,0])

    elif metric == "specificity":
     return cm[0,0] / float(cm[0,0] + cm[0,1])
```

```
    return 0
    ### ========== TODO : END ========== ###


def test_performance() :
    # np.random.seed(1234)
    # y_true = 2 * np.random.randint(0,2,10) - 1
    # np.random.seed(2345)
    # y_pred = (10 + 10) * np.random.random(10) - 10

    y_true = [ 1,  1, -1,  1, -1, -1, -1,  1,  1,  1]
   #y_pred = [ 1, -1,  1, -1,  1,  1, -1, -1,  1, -1]
   # confusion matrix
   #          pred pos     neg
   # true pos      tp (2)  fn (4)
   #      neg      fp (3)  tn (1)
    y_pred = [ 3.21288618, -1.72798696,  3.36205116, -5.40113156,  6.15356672,
               2.73636929, -6.55612296, -4.79228264,  8.30639981, -0.74368981]
    metrics = ["accuracy", "f1_score", "auroc", "precision", "sensitivity", "specificity"]
    scores  = [     3/10.,      4/11.,   5/12.,        2/5.,          2/6.,          1/4.]

    import sys
    eps = sys.float_info.epsilon

    for i, metric in enumerate(metrics) :
        assert abs(performance(y_true, y_pred, metric) - scores[i]) < eps, \
            (metric, performance(y_true, y_pred, metric), scores[i])


def cv_performance(clf, X, y, kf, metric="accuracy"):
    """
    Splits the data, X and y, into k-folds and runs k-fold cross-validation.
    Trains classifier on k-1 folds and tests on the remaining fold.
    Calculates the k-fold cross-validation performance metric for classifier
    by averaging the performance across folds.

    Parameters
    --------------------
        clf    -- classifier (instance of SVC)
        X      -- numpy array of shape (n,d), feature vectors
                    n = number of examples
                    d = number of features
        y      -- numpy array of shape (n,), binary labels {1,-1}
        kf     -- cross_validation.KFold or cross_validation.StratifiedKFold
        metric -- string, option used to select performance measure

    Returns
    --------------------
```

```
        score   -- float, average cross-validation performance across k folds
    """

    scores = []
    for train, test in kf :
        X_train, X_test, y_train, y_test = X[train], X[test], y[train], y[test]
        clf.fit(X_train, y_train)
        # use SVC.decision_function to make ``continuous-valued'' predictions
        y_pred = clf.decision_function(X_test)
        score = performance(y_test, y_pred, metric)
        if not np.isnan(score) :
            scores.append(score)
    return np.array(scores).mean()


def select_param_linear(X, y, kf, metric="accuracy"):
    """
    Sweeps different settings for the hyperparameter of a linear-kernel SVM,
    calculating the k-fold CV performance for each setting, then selecting the
    hyperparameter that 'maximize' the average k-fold CV performance.

    Parameters
    --------------------
        X      -- numpy array of shape (n,d), feature vectors
                    n = number of examples
                    d = number of features
        y      -- numpy array of shape (n,), binary labels {1,-1}
        kf     -- cross_validation.KFold or cross_validation.StratifiedKFold
        metric -- string, option used to select performance measure

    Returns
    --------------------
        C -- float, optimal parameter value for linear-kernel SVM
    """

    print 'Linear SVM Hyperparameter Selection based on ' + str(metric) + ':'
    C_range = 10.0 ** np.arange(-3, 3)

    ### ========== TODO : START ========== ###
    # part 2c: select optimal hyperparameter using cross-validation
    bestC = None
    bestPerf = 0
    for c in C_range:
     clf = SVC(C=c, kernel="linear")
     perf = cv_performance(clf, X, y, kf, metric)
     if perf > bestPerf:
     bestPerf = perf
     bestC = c
```

```
    return bestC
    ### ========== TODO : END ========== ###


def select_param_rbf(X, y, kf, metric="accuracy"):
    """
    Sweeps different settings for the hyperparameters of an RBF-kernel SVM,
    calculating the k-fold CV performance for each setting, then selecting the
    hyperparameters that 'maximize' the average k-fold CV performance.

    Parameters
    --------------------
        X       -- numpy array of shape (n,d), feature vectors
                     n = number of examples
                     d = number of features
        y       -- numpy array of shape (n,), binary labels {1,-1}
        kf     -- cross_validation.KFold or cross_validation.StratifiedKFold
        metric  -- string, option used to select performance measure

    Returns
    --------------------
        gamma, C -- tuple of floats, optimal parameter values for an RBF-kernel SVM
    """

    print 'RBF SVM Hyperparameter Selection based on ' + str(metric) + ':'

    ### ========== TODO : START ========== ###
    # part 3b: create grid, then select optimal hyperparameters using cross-validation
    CList = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000.0, 10000.0, 1000000.0]
    gammaList = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000.0, 10000.0, 1000000.0]

    bestC = 0
    bestGamma = 0
    bestPerf = 0
    for c in CList:
        for gamma in gammaList:
            clf = SVC(C=c, gamma=gamma, kernel="rbf")
            perf = cv_performance(clf, X, y, kf, metric)
            if perf > bestPerf:
                bestPerf = perf
                bestC = c
                bestGamma = gamma
    print "Performance for " + str(metric) + " is " + str(bestPerf)
    return bestGamma, bestC
    ### ========== TODO : END ========== ###

def shorterParamSelect(metric_list):
    """
```

```
Sweeps different settings for the hyperparameters of an SVM
calculating the k-fold CV performance for each setting, then selecting the
hyperparameters that 'maximize' the average k-fold CV performance.

Parameters
--------------------
    metric_list       -- list of metrics to try

Returns
--------------------
    Nothing
"""

tweetList = []

with open('../data/tweets.txt', 'rU') as fid :
    for  line in fid:
        tweetList.append(line)
yBase = read_vector_file('../data/labels.txt')

CList = [10.0, 100.0, 1000.0, 10000.0]
gammaList = [0.0001, 0.001, 0.01]

# Search over what was found to be the best for accuracy
# For each comparison metric
cv = CountVectorizer(max_features=500)
cv.fit(tweetList)
dtm = cv.transform(tweetList)
dtm, y = shuffle(dtm, yBase, random_state=0)
dtm_train, dtm_test = dtm[:560], dtm[560:]
y_train, y_test = y[:560], y[560:]
kf = StratifiedKFold(y_train, 5)
clf = SVC(C=10., gamma=0.001, kernel="rbf")
clf.fit(dtm_train, y_train)
for metric in metric_list:
    print "Metric: " + metric
    print cv_performance(clf, dtm_train, y_train, kf, metric=metric)
    print performance_CI(clf, dtm_test, y_test, metric=metric)

# Search over all parameters
bestC = 0
bestGamma = 0
bestFeat = 0
bestPerf = 0
bestKernel = None
bestTransform = False
for numFeat in range(500, 1500, 100):
        for c in CList:
```

```
                for gamma in gammaList:
                    for kernelType in ["linear", "rbf"]:
                        for useTFIDF in [True, False]:
                            cv = CountVectorizer(max_features=numFeat)
                            cv.fit(tweetList)
                            dtm = cv.transform(tweetList)
                            if useTFIDF:
                                tfidf = TfidfTransformer()
                                dtm = tfidf.fit_transform(dtm)
                            dtm, y = shuffle(dtm, yBase, random_state=0)
                            dtm_train, dtm_test = dtm[:560], dtm[560:]
                            y_train, y_test = y[:560], y[560:]
                            kf = StratifiedKFold(y_train, 5)
                            clf = SVC(C=c, gamma=gamma, kernel=kernelType)
                            clf.fit(dtm_train, y_train)
                            perf, _, _ = performance_CI(clf, dtm_test, y_test, metric="accuracy
                            if perf > bestPerf:
                                bestPerf = perf
                                bestC = c
                                bestFeat = numFeat
                                bestGamma = gamma
                                bestKernel = kernelType
                                bestTransform = useTFIDF
                                print "Perf: " + str(bestPerf)
                                print "C: " + str(bestC)
                                print "Gamma: " + str(bestGamma)
                                print "Feat: " + str(bestFeat)
                                print "Kernel: " + str(bestKernel)
                                print "Transform: " + str(bestTransform)
    print "Performance for " + str(metric) + " is " + str(bestPerf)


def performance_CI(clf, X, y, metric="accuracy"):
    """
    Estimates the performance of the classifier using the 95% CI.

    Parameters
    --------------------
        clf          -- classifier (instance of SVC)
                          [already fit to data]
        X            -- numpy array of shape (n,d), feature vectors of test set
                          n = number of examples
                          d = number of features
        y            -- numpy array of shape (n,), binary labels {1,-1} of test set
        metric       -- string, option used to select performance measure

    Returns
    --------------------
```

```
        score        -- float, classifier performance
        lower, upper -- tuple of floats, confidence interval
    """

    y_pred = clf.decision_function(X)
    score = performance(y, y_pred, metric)

    ### ========== TODO : START ========== ###
    # part 4b: use bootstrapping to compute 95% confidence interval
    # hint: use np.random.randint(...)
    n, d = X.shape

    perf = []

    for t in range(1000):
        indices = np.random.randint(0, n, size = n)

        X_bootstrap = X[indices]
        y_bootstrap = y[indices]

        y_pred = clf.decision_function(X_bootstrap)
        perf.append(performance(y_bootstrap, y_pred, metric=metric))

    lower = np.percentile(perf, 2.5)
    upper = np.percentile(perf, 97.5)

    return score, lower, upper
    ### ========== TODO : END ========== ###


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

def main() :
    # read the tweets and its labels
    dictionary = extract_dictionary('../data/tweets.txt')
    test_extract_dictionary(dictionary)
    X = extract_feature_vectors('../data/tweets.txt', dictionary)
    test_extract_feature_vectors(X)
    y = read_vector_file('../data/labels.txt')

    # shuffle data (since file has tweets ordered by movie)
    X, y = shuffle(X, y, random_state=0)

    # set random seed
    np.random.seed(1234)
```

```
# split the data into training (training + cross-validation) and testing set
X_train, X_test = X[:560], X[560:]
y_train, y_test = y[:560], y[560:]

metric_list = ["accuracy", "f1_score", "auroc", "precision", "sensitivity", "specificity"]

### ========== TODO : START ========== ###
test_performance()

# part 2b: create stratified folds (5-fold CV)
kf = StratifiedKFold(y_train, 5)

# part 2d: for each metric, select optimal hyperparameter for linear-kernel SVM using CV
print "Performance across models and C values is..."
perf = []
for c in 10.0 ** np.arange(-3, 3):
 innerPerf = []
 for metric in metric_list:
 clf = SVC(C=c, kernel="linear")
 innerPerf.append(round(cv_performance(clf, X_train, y_train, kf, metric), 4))
 perf.append(innerPerf)
 print innerPerf
print np.max(np.asarray(perf), axis=0)

# part 3c: for each metric, select optimal hyperparameter for RBF-SVM using CV
print "Performance across models and C values is..."
perf = []
for metric in metric_list:
    gamma, c = select_param_rbf(X_train, y_train, kf, metric=metric)
    print "Metric: " + str(metric) + " gamma: " + str(gamma) + " c " + str(c)


# part 4a: train linear- and RBF-kernel SVMs with selected hyperparameters
c = 1.
linClf = SVC(C=c, kernel="linear")
c = 100.0
gamma = 0.01
rbfClf = SVC(C=c, gamma=gamma, kernel="rbf")
linClf.fit(X_train, y_train)
rbfClf.fit(X_train, y_train)

# part 4c: use bootstrapping to report performance on test data
for metric in metric_list:
    score1, lower1, upper1 = performance_CI(linClf, X_test, y_test, metric=metric)
    score2, lower2, upper2 = performance_CI(rbfClf, X_test, y_test, metric=metric)
    print "Metric: " + str(metric)
    print "Linear: Score: " + str(score1) + " lower: " + str(lower1) + " upper: " + str(up
    print "RBF: Score: " + str(score2) + " lower: " + str(lower2) + " upper: " + str(upper
```

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

### ========== TODO : START ========== ###
# Twitter contest
# uncomment out the following, and be sure to change the filename

# Learning curve plot
n, d = np.shape(X_train)
perf = []
perfTrain = []
percentage = []
for i in xrange(10, 81, 10):
 percentage.append(i)
 linClf = SVC(C=1, kernel="linear")
 linClf.fit(X_train[:int((i/100.0)*n)], y_train[:int((i/100.0)*n)])
 y_true = y_train[int((i/100.0)*n):]
 y_pred = linClf.decision_function(X_train[int((i/100.0)*n):])
 perf.append(1-performance(y_true, y_pred, "accuracy"))
 y_test = y_train[:int((i/100.0)*n)]
 y_pred = linClf.decision_function(X_train[:int((i/100.0)*n)])
 perfTrain.append(1-performance(y_test, y_pred, "accuracy"))
matplotlib.pyplot.plot(np.asarray(percentage), np.asarray(perf),
    c='b', label='Test Error')
matplotlib.pyplot.plot(np.asarray(percentage), np.asarray(perfTrain),
    c='g', label='Train Error')
plt.autoscale(enable=True)
plt.xlabel('percentage data')
plt.ylabel('error')
plt.legend(loc=1,prop={'size':8})
plt.show()

# Code used to find good parameters - see function above
shorterParamSelect(metric_list)

# Read file into list of tweets
tweetList = []
with open('../data/tweets.txt', 'rU') as fid :
    for  line in fid:
        tweetList.append(line)

cv = CountVectorizer(max_features=500)
cv.fit(tweetList)
dtm = cv.transform(tweetList)
y = read_vector_file('../data/labels.txt')
dtm, y = shuffle(dtm, y, random_state=0)
clf = SVC(C=10.0, gamma=0.001, kernel="rbf")
clf.fit(dtm, y)
```

```
    # Bring in held out tweet data
    heldOutTweetList = []
    with open('../data/held_out_tweets.txt', 'rU') as fid :
        for  line in fid:
            heldOutTweetList.append(line)
    heldOutDTM = cv.transform(heldOutTweetList)
    y_pred = clf.decision_function(heldOutDTM)
    write_label_answer(y_pred, '../data/sbaron_vkishore_twitter.txt')
    ### ========== TODO : END ========== ###


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