03/04/18 03:45:11
/Users/caiglencross/Documents/MachineLearning/ps2/ps6/source/twitter.py

```python
"""
Author      : Cai Glencross
Class       : HMC CS 158
Date        : 2018 Feb 14
Description : Twitter
"""

"""
Author: Cai Glencross, Katie Li
"""

from string import punctuation

# numpy libraries
import numpy as np

# matplotlib libraries
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt

# scikit-learn libraries
from sklearn.dummy import DummyClassifier
from sklearn.svm import SVC
from sklearn.model_selection import StratifiedKFold
from sklearn import metrics
from sklearn.utils import shuffle

######################################################################
# 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()
```

```python


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
        index = 0
        for input_string in fid:
            words = extract_words(input_string)

            for word in words:
                if (not (word in word_list)):
                    word_list[word] = index
                    index = index + 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
        n = 0 # set index of line number
        for input_string in fid:
            words = extract_words(input_string)
            for word in words:
                index = word_list[word]
                feature_matrix[n, index] = 1
            n = n + 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

    tn, fp, fn, tp = metrics.confusion_matrix(y_true, y_label).ravel()

    if (metric == "accuracy"):
        accuracy = metrics.accuracy_score(y_true, y_label, normalize = True)
        selected_metric = accuracy
    elif (metric == "f1_score"):
        f1score = metrics.f1_score(y_true, y_label)
        selected_metric = f1score
    elif (metric == "precision"):
        precision = metrics.precision_score(y_true, y_label)
        selected_metric = precision
    elif (metric == "auroc"):
        auroc = metrics.roc_auc_score(y_true, y_pred)
        selected_metric = auroc
    elif (metric == "sensitivity"):
        sensitivity = float(tp)/(tp + fn)
        selected_metric = sensitivity
    elif (metric == "specificity"):
        specificity =float(tn) /( tn + fp)
        selected_metric = specificity

    return selected_metric
    ### ========== 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     -- model_selection.KFold or model_selection.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.split(X, y) :
        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", plot=True) :
    """
    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     -- model_selection.KFold or model_selection.StratifiedKFold
        metric -- string, option used to select performance measure
        plot   -- boolean, make a plot

    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
    best_c = (-1,-1) # (score, C_best)
    scores = []
    for c_i in C_range:
        svm = SVC(C = c_i, kernel ='linear')
        score = cv_performance(svm, X, y, kf, metric=metric)
        scores.append(score)
        if (best_c[0] == -1 ) or (score > best_c[0]):
            best_c = (score, c_i)

    if plot:
        lineplot(C_range, scores, metric)
        plt.hold()

```

```
    print(best_c[1]) # print out the optimal hyperparameter score
    return best_c[1]
    ### ========== TODO : END ========== ###


def plot_metric_2d(X, y, kf) :
    """
    Plots line plots of all the metrics

    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      -- model_selection.KFold or model_selection.StratifiedKFold

    Action
    --------------------
        creates a line plot
    """

    C_range = 10.0 ** np.arange(-3, 3)
    metric_list = ["accuracy", "f1_score", "auroc", "precision", "sensitivity", "specificity"]

    ### ========== TODO : START ========== ###
    # part 2c: select optimal hyperparameter using cross-validation
    for metric in metric_list:
        scores = []
        for c_i in C_range:
            svm = SVC(C = c_i, kernel ='linear')
            score = cv_performance(svm, X, y, kf, metric=metric)
            scores.append(score)

        xx = range(len(scores))
        plt.plot(xx, scores, linestyle='-', linewidth=2, label=metric)
        plt.xticks(xx, C_range)
        plt.xlabel("C")
        plt.ylabel("Scores")
        plt.title("Classifier Performance")
        plt.legend()

    plt.show()

    ### ========== 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      -- model_selection.KFold or model_selection.StratifiedKFold
        metric  -- string, option used to select performance measure

    Returns
    --------------------
        C        -- float, optimal parameter value for an RBF-kernel SVM
        gamma    -- float, optimal parameter value 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

    #rows are gamma and columns are C
    performance_grid = np.zeros((5,4))
    C_vals = [0.01, 0.1, 1.0, 10.0, 100.0, 1000.0]
    gamma_vals = [0.001, 0.01, 0.1,1.0, 10]
    for i in range(0,5):
        for j in range(0,4):
            performance_grid[i][j] = cv_performance(SVC(kernel='rbf', gamma=gamma_vals[i],C=C_vals[j]),
                                                    X, y, kf, metric=metric)

    indices = np.unravel_index(np.argmax(performance_grid), performance_grid.shape)

    return C_vals[indices[1]], gamma_vals[indices[0]], performance_grid[indices]
    ### ========== TODO : END ========== ###
```

```


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

    Parameters
    --------------------
        clf          -- classifier (instance of SVC or DummyClassifier)
                          [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        -- float, lower limit of confidence interval
        upper        -- float, upper limit of confidence interval
    """
    n, d = X.shape
    try :
        y_pred = clf.decision_function(X)
    except :
        y_pred = clf.predict(X)
    score = performance(y, y_pred, metric)

    ### ========== TODO : START ========== ###
    # part 4b: use bootstrapping to compute 95% confidence interval
    # hint: use np.random.randint(...)
    confidence_array = []
    for t in range(0,1000):
        bootstrapped_X = np.zeros((n,d))
        bootstrapped_y = np.zeros(n)
        bootstrapped_ypred = np.zeros(n)
        avg = 0
        for i in range(0,n):
            index = np.random.randint(0, n)
            bootstrapped_X[i,:]=X[index,:]
            bootstrapped_y[i] = y[index]
            bootstrapped_ypred[i] = y_pred[index]
        confidence_array.append(performance(bootstrapped_y, bootstrapped_ypred, metric))

```

```

    confidence_array.sort()

    return score, confidence_array[24], confidence_array[974]
    ### ========== TODO : END ========== ###


######################################################################
# functions -- plotting
######################################################################

def lineplot(x, y, label):
    """
    Make a line plot.

    Parameters
    --------------------
        x            -- list of doubles, x values
        y            -- list of doubles, y values
        label        -- string, label for legend
    """

    xx = range(len(x))
    plt.plot(xx, y, linestyle='-', linewidth=2, label=label)
    plt.xticks(xx, x)
    plt.show()


def plot_results(metrics, classifiers, *args):
    """
    Make a results plot.

    Parameters
    --------------------
        metrics      -- list of strings, metrics
        classifiers  -- list of strings, classifiers
        args         -- variable length argument
                          results for baseline
                          results for classifier 1
                          results for classifier 2
                          ...
                        each results is a tuple (score, lower, upper)
    """

    num_metrics = len(metrics)
    num_classifiers = len(args) - 1

```

```
    ind = np.arange(num_metrics)  # the x locations for the groups
    width = 0.7 / num_classifiers # the width of the bars

    fig, ax = plt.subplots()

    # loop through classifiers
    rects_list = []
    for i in xrange(num_classifiers):
        results = args[i+1] # skip baseline
        means = [it[0] for it in results]
        errs = [(it[0] - it[1], it[2] - it[0]) for it in results]
        rects = ax.bar(ind + i * width, means, width, label=classifiers[i])
        ax.errorbar(ind + i * width, means, yerr=np.array(errs).T, fmt='none', ecolor='k')
        rects_list.append(rects)

    # baseline
    results = args[0]
    for i in xrange(num_metrics) :
        mean = results[i][0]
        err_low = results[i][1]
        err_high = results[i][2]
        xlim = (ind[i] - 0.8 * width, ind[i] + num_classifiers * width - 0.2 * width)
        plt.plot(xlim, [mean, mean], color='k', linestyle='-', linewidth=2)
        plt.plot(xlim, [err_low, err_low], color='k', linestyle='--', linewidth=2)
        plt.plot(xlim, [err_high, err_high], color='k', linestyle='--', linewidth=2)

    ax.set_ylabel('Score')
    ax.set_ylim(0, 1)
    ax.set_xticks(ind + width / num_classifiers)
    ax.set_xticklabels(metrics)
    ax.legend()

    def autolabel(rects):
        """Attach a text label above each bar displaying its height"""
        for rect in rects:
            height = rect.get_height()
            ax.text(rect.get_x() + rect.get_width()/2., 1.05*height,
                    '%.3f' % height, ha='center', va='bottom')

    for rects in rects_list:
        autolabel(rects)

    plt.show()
```

```


######################################################################
# 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_strat = StratifiedKFold(n_splits=5)
    cv_scores = cv_performance(SVC(), X_train, y_train, kf_strat)
    print "scores for CV: " + str(cv_scores)

    # part 2c: finding the optimal C
    best_cs = []
    for metric in metric_list:
        best_cs.append(select_param_linear(X, y, kf_strat, metric=metric, plot=False))
    print best_cs
    # part 2d: for each metric, select optimal hyperparameter for linear-kernel SVM using CV
    # plot the metrics
    plot_metric_2d(X, y, kf_strat)

    # part 3c: for each metric, select o
```

```
    # optimal hyperparameter for RBF-SVM using CV
    C, gamma, score = select_param_rbf(X, y, kf_strat)
    print "optimal C for accuracy= %f, optimal gamma for accuracy %f, score was %f" % (C, gamma, score)



    metrics = ["accuracy", "f1_score", "auroc", "precision", "sensitivity", "specificity"]
    for metric in metrics:
        #C, gamma, score = select_param_rbf(X, y, kf_strat, metric=metric)
        score = cv_performance(SVC(kernel='rbf', gamma=0.01, C=10.0),
                                                X, y, kf_strat, metric=metric)
        print "optimal C for %s= %f, optimal gamma %f, score was %f" % (metric, 10.0, 0.01, score)





    # part 4a: train linear- and RBF-kernel SVMs with selected hyperparameters
    linear_svm = SVC(kernel='linear', C=1)
    rbf_svm = SVC(kernel='rbf', gamma=0.01, C=10.0)
    dummy_classifier = DummyClassifier(strategy="most_frequent")
    linear_svm.fit(X_train,y_train)
    rbf_svm.fit(X_train,y_train)
    dummy_classifier.fit(X_train, y_train)

    # part 4c: use bootstrapping to report performance on test data
             # use plot_results(...) to make plot
    linear_svm_performance = []
    print "LINEAR SVM"
    for metric in metrics:
        result_metric = performance_CI(linear_svm, X_test, y_test, metric= metric)
        linear_svm_performance.append(result_metric)

        print "for %s: score = %f, low end: %f, high end = %f" % (metric, result_metric[0],
                                                                  result_metric[1], result_metric[2])

    rbf_svm_performance = []
    print "RBF SVM"
    for metric in metrics:
        result_metric = performance_CI(rbf_svm, X_test, y_test,
```

```
metric= metric)
        rbf_svm_performance.append(result_metric)
        print "for %s: score = %f, low end: %f, high end = %f" %
(metric, result_metric[0],

result_metric[1], result_metric[2])

    # use a baseline performance classifier for comparison
    dummy_classifier_performance = []
    print "Baseline classifier - majority classifier with Dummy
Classifier"
    for metric in metrics:
        result_metric = performance_CI(dummy_classifier, X_test,
y_test, metric= metric)
        dummy_classifier_performance.append(result_metric)
        print "for %s: score = %f, low end: %f, high end = %f" %
(metric, result_metric[0],

result_metric[1], result_metric[2])



    # create the bar plot
    classifiers = ["Linear SVM", "RBF SVM"]
    plot_results(metrics, classifiers, dummy_classifier_performance,
linear_svm_performance, rbf_svm_performance)


    # part 5: identify important features
    full_linear_svm = SVC(kernel='linear', C=1)
    full_linear_svm.fit(X,y)

    print "positive linear coefs: "

    coefs_index = full_linear_svm.coef_[0].argsort()[-10:][::-1]
    for coef in coefs_index:
        print full_linear_svm.coef_[0][coef]
        for word, index in dictionary.iteritems():
            if index == coef:
                print word



    print "negative linear coefs: "
    coefs_index = full_linear_svm.coef_[0].argsort()[:10][::-1]
    for coef in coefs_index:
        print full_linear_svm.coef_[0][coef]
        for word, index in dictionary.iteritems():
            if index == coef:
                print word
```

```

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

    ### ========== TODO : START ========== ###
    # Twitter contest
    # uncomment out the following, and be sure to change the filename
    """
    X_held = extract_feature_vectors('../data/held_out_tweets.txt', dictionary)
    # your code here
    # y_pred = best_clf.decision_function(X_held)
    write_label_answer(y_pred, '../data/yjw_twitter.txt')
    """
    ### ========== TODO : END ========== ###


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