```
import os
import sys


# To add your own Drive Run this cell.
from google.colab import drive
drive.mount('/content/drive')
```

```
    Mounted at /content/drive
```

```
# Please append your own directory after '/content/drive/My Drive/'
### ========== TODO : START ========== ###
sys.path += ['/content/drive/My Drive/hw3_code/src']
### ========== TODO : END ========== ###


"""
Author      : Yi-Chieh Wu, Sriram Sankararman
Description : Twitter
"""

from string import punctuation

import numpy as np
import matplotlib.pyplot as plt
# !!! MAKE SURE TO USE LinearSVC.decision_function(X), NOT LinearSVC.predict(X) !!!
# (this makes ''continuous-valued'' predictions)
from sklearn.svm import LinearSVC
from sklearn.model_selection import StratifiedKFold
from sklearn import metrics
```

Problem 3: Twitter Analysis Using SVM

```
######################################################################
# 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 = {}
    idx = 0
    with open(infile, 'r') as fid :
        # process each line to populate word_list
        for input_string in fid:
            words = extract_words(input_string)
            for word in words:
                if word not in word_list:
                    word_list[word] = idx
                    idx += 1
    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,'r'))
    num_words = len(word_list)
    feature_matrix = np.zeros((num_lines, num_words))

    with open(infile, 'r') as fid :
        # process each line to populate feature_matrix
        for i, input_string in enumerate(fid):
            words = extract_words(input_string)
            for word in words:
```

```
                feature_matrix[i, word_list[word]] = 1.0

    return feature_matrix
```

```
######################################################################
# functions -- evaluation
######################################################################
from sklearn.metrics import accuracy_score, f1_score, roc_auc_score, precision_score, recall_score, confusion_matrix

def performance(y_true, y_pred, metric):
    """
    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

    ### ========== TODO : START ========== ###
    # part 1a: compute classifier performance
    if metric == "accuracy":
      score = accuracy_score(y_true, y_label)
    elif metric == "f1-score":
      score = f1_score(y_true, y_label)
    elif metric == "auroc":
      score = roc_auc_score(y_true, y_label)
    elif metric == "precision":
      score = precision_score(y_true, y_label)
    elif metric == "sensitivity":
      score = recall_score(y_true, y_label)
    elif metric == "specificity":
      temp = confusion_matrix(y_true, y_label)
      score = temp[0,0] / (temp[0,0]+temp[0,1])

    return score
    ### ========== TODO : END ========== ###


def cv_performance(clf, X, y, kf, metric):
    """
    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 LinearSVC)
        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.StratifiedKFold
        metric -- string, option used to select performance measure

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

    ### ========== TODO : START ========== ###
    # part 1b: compute average cross-validation performance
    scores = []

    for train_i, test_i in kf.split(X, y):
      X_train, y_train = X[train_i], y[train_i]
      X_test, y_test = X[test_i], y[test_i]

      clf.fit(X_train, y_train)
```

```
      y_pred = clf.decision_function(X_test)
      score1 = performance(y_test, y_pred, metric)

      scores.append(score1)

    score = np.mean(scores)

    return score

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


def select_param_linear(X, y, kf, metric):
    """
    Sweeps different settings for the hyperparameter of a linear 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.StratifiedKFold
        metric -- string, option used to select performance measure

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

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

    ### ========== TODO : START ========== ###
    # part 1c: select optimal hyperparameter using cross-validation
    opt_c = None
    bscore = -1


    for c in C_range:
      clf = LinearSVC(loss='hinge',random_state=0, C=c)
      a_score = cv_performance(clf, X, y, kf, metric)

      if a_score > bscore:
        bscore = a_score
        opt_c = c

    return opt_c
    ### ========== TODO : END ========== ###


def performance_test(clf, X, y, metric):
    """
    Estimates the performance of the classifier.

    Parameters
    --------------------
        clf          -- classifier (instance of LinearSVC)
                          [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
    """


    ### ========== TODO : START ========== ###
    # part 2b: return performance on test data under a metric.
    y_pred = clf.predict(X)
```

```
if metric == "accuracy":
  score = accuracy_score(y, y_pred)
elif metric == "f1-score":
  score = f1_score(y, y_pred)
elif metric == "auroc":
  score = roc_auc_score(y, y_pred)
elif metric == "precision":
  score = precision_score(y, y_pred)
elif metric == "sensitivity":
  score = recall_score(y, y_pred)
elif metric == "specificity":
  temp = confusion_matrix(y, y_pred)
  score = temp[0,0] / (temp[0,0]+temp[0,1])

return score


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

```
######################################################################
# main
######################################################################

def main() :
    np.random.seed(1234)

    # read the tweets and its labels, change the following two lines to your own path.
    ### ========== TODO : START ========== ###
    file_path = '/content/drive/My Drive/hw3_code/data/tweets.txt'
    label_path = '/content/drive/My Drive/hw3_code/data/labels.txt'
    ### ========== TODO : END ========== ###
    dictionary = extract_dictionary(file_path)
    print(len(dictionary))
    X = extract_feature_vectors(file_path, dictionary)
    y = read_vector_file(label_path)
    # split 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 ========== ###
    # part 1b: create stratified folds (5-fold CV)
    stkf = StratifiedKFold(n_splits=5, shuffle=True, random_state=None)

    for train_i, test_i in stkf.split(X_train, y_train):
      Xtrain, Xtest = X[train_i], X[test_i]
      ytrain, ytest = y[train_i], y[test_i]

    # part 1c: for each metric, select optimal hyperparameter for linear SVM using CV
    for m in metric_list:
      my_c = select_param_linear(X_train, y_train, stkf, m)
      print(str(my_c))

    # part 2a: train linear SVMs with selected hyperparameters
    c_ac = select_param_linear(X_train, y_train, stkf, 'accuracy')
    c_f = select_param_linear(X_train, y_train, stkf, 'f1-score')
    c_au = select_param_linear(X_train, y_train, stkf, 'auroc')
    c_p = select_param_linear(X_train, y_train, stkf, 'precision')
    c_se = select_param_linear(X_train, y_train, stkf, 'sensitivity')
    c_sp = select_param_linear(X_train, y_train, stkf, 'specificity')

    svm_ac = LinearSVC(loss='hinge', random_state=0, C=c_ac)
    svm_f = LinearSVC(loss='hinge', random_state=0, C=c_f)
    svm_au = LinearSVC(loss='hinge', random_state=0, C=c_au)
    svm_p = LinearSVC(loss='hinge', random_state=0, C=c_p)
    svm_se = LinearSVC(loss='hinge', random_state=0, C=c_se)
    svm_sp = LinearSVC(loss='hinge', random_state=0, C=c_sp)

    svm_ac.fit(X_train, y_train)
    svm_f.fit(X_train, y_train)
    svm_au.fit(X_train, y_train)
    svm_p.fit(X_train, y_train)
    svm_se.fit(X_train, y_train)
    svm_sp.fit(X_train, y_train)

    # part 2b: test the performance of your classifiers.
    ac_s = performance_test(svm_ac, X_test, y_test, 'accuracy')
    f_s = performance_test(svm_f, X_test, y_test, 'f1-score')
    au_s = performance_test(svm_au, X_test, y_test, 'auroc')
    p_s = performance_test(svm_ac, X_test, y_test, 'precision')
    se_s = performance_test(svm_ac, X_test, y_test, 'sensitivity')
    sp_s = performance_test(svm_ac, X_test, y_test, 'specificity')

    print("Accuracy Score: " + str(ac_s))
    print("F1 Score: " + str(f_s))
    print("Auroc Score: " + str(au_s))
    print("Precision Score: " + str(p_s))
    print("Sensitivity Score: " + str(se_s))
    print("Specificity Score: " + str(sp_s))

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


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

Problem 4: Boosting vs. Decision Tree

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn import metrics
from sklearn.model_selection import cross_val_score, train_test_split
```

```
class Data :

    def __init__(self) :
        """
        Data class.

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

        # n = number of examples, d = dimensionality
        self.X = None
        self.y = None

        self.Xnames = None
        self.yname = None
```