01/30/18 10:11:36 /Users/caiglencross/Documents/MachineLearning/ps2/source/titanic.py

```
"""
Author      : Cai Glencross and Katie Li
Class       : HMC CS 158
Date        : 2018 Jan 30
Description : Titanic
"""

# Use only the provided packages!
import math
import csv
from util import *

from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn import metrics

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

class Classifier(object) :
    """
    Classifier interface.
    """

    def fit(self, X, y):
        raise NotImplementedError()

    def predict(self, X):
        raise NotImplementedError()


class MajorityVoteClassifier(Classifier) :

    def __init__(self) :
        """
        A classifier that always predicts the majority class.

        Attributes
        --------------------
            prediction_ -- majority class
        """
        self.prediction_ = None

    def fit(self, X, y) :
        """
        Build a majority vote classifier from the training set (X, y).

        Parameters
        --------------------
            X    -- numpy array of shape (n,d), samples
            y    -- numpy array of shape (n,), target classes

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

```python
        vals, counts = np.unique(y, return_counts=True)
        majority_val, majority_count = max(zip(vals, counts), key=lambda (val, count): count)
        self.prediction_ = majority_val
        return self

    def predict(self, X) :
        """
        Predict class values.

        Parameters
        --------------------
            X    -- numpy array of shape (n,d), samples

        Returns
        --------------------
            y    -- numpy array of shape (n,), predicted classes
        """
        if self.prediction_ is None :
            raise Exception("Classifier not initialized. Perform a fit first.")

        n,d = X.shape
        y = [self.prediction_] * n
        return y


class RandomClassifier(Classifier) :

    def __init__(self) :
        """
        A classifier that predicts according to the distribution of the classes.

        Attributes
        --------------------
            probabilities_ -- class distribution dict (key = class, val = probability of class)
        """
        self.probabilities_ = None

    def fit(self, X, y) :
        """
        Build a random classifier from the training set (X, y).

        Parameters
        --------------------
            X    -- numpy array of shape (n,d), samples
            y    -- numpy array of shape (n,), target classes

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

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

        # part c: set self.probabilities_ according to the training set

        vals, counts = np.unique(y, return_counts=True)
        majority_val, majority_count = max(zip(vals, counts), key=lambda (val, count): count)
        minority_val, minority_count = min(zip(vals, counts), key=lambda (val, count): count)

        # find the highest probability
        self.probabilities_ = {majority_val: float(majority_count)/sum(counts),
                               minority_val: float(minority_count)/sum(counts)}

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

        return self

    def predict(self, X, seed=1234) :
        """
        Predict class values.

        Parameters
        --------------------
            X    -- numpy array of shape (n,d), samples
            seed -- integer, random seed

        Returns
        --------------------
            y    -- numpy array of shape (n,), predicted classes
        """
        if self.probabilities_ is None :
            raise Exception("Classifier not initialized. Perform a fit first.")
        np.random.seed(seed)

        ### ========== TODO : START ========== ###
        # part c: predict the class for each test example
        # hint: use np.random.choice (be careful of the parameters)

        class_distr = self.probabilities_
        n, d = X.shape

        # Create a random distribution based on given probabilities of survival
        random_ys = np.random.choice(class_distr.keys(), n, p = class_distr.values())
        y = random_ys
        ### ========== TODO : END ========== ###

        return y


######################################################################
# functions
######################################################################

def error(clf, X, y, ntrials=100, test_size=0.2) :
    """
    Computes the classifier error over a random split of the data,
    averaged over ntrials runs.

    Parameters
    --------------------
        clf         -- classifier
        X           -- numpy array of shape (n,d), features values
        y           -- numpy array of shape (n,), target classes
        ntrials     -- integer, number of trials
        test_size   -- float (between 0.0 and 1.0) or int,
                       if float, the proportion of the dataset to include in the test split
                       if int, the absolute number of test samples

    Returns
    --------------------
        train_error -- float, training error
        test_error  -- float, test error
    """

    ### ========== TODO : START ========== ###
    # part b: compute cross-validation error over ntrials
    # hint: use train_test_split (be careful of the parameters)
    SEED = 1234
    np.random.seed(SEED)

    train_error = 0
    test_error = 0

    test_error_total = 0
    train_error_total = 0

    n, d = X.shape



    for t in range (0, ntrials):
        # get the split of data
        X_train = np.empty((0,d)); X_test = np.empty((0,d))
        y_train = []; y_test = []

        for i in range(0, n):
            if (np.random.random_sample() < (1.0 - test_size)):
                X_train = np.vstack((X_train, X[i,:]))
                y_train.append(y[i])
            else:
                X_test = np.vstack((X_test,X[i,:]))
                y_test.append(y[i])

        # print "X_train shape is", X_train.shape
        # print "shape of first element of X_train is:", X[0,:].shape
        # print "shape of second elt:", X[1,:].shape
        # print "shape of stack is:", np.vstack((X[0,:],X[1,:])).shape
        # print "y train length is:", len(y)
        clf.fit(X_train,y_train)
        y_pred_test = clf.predict(X_test)
        y_pred_train = clf.predict(X_train)
        test_error_total += 1 - metrics.accuracy_score(y_pred_test, y_test, normalize=True)
        train_error_total += 1 - metrics.accuracy_score(y_pred_train, y_train, normalize=True)

    train_error = train_error_total/ntrials
    test_error = test_error_total/ntrials
    ### ========== TODO : END ========== ###

    return train_error, test_error


def write_predictions(y_pred, filename, yname=None) :
    """Write out predictions to csv file."""
    out = open(filename, 'wb')
    f = csv.writer(out)
    if yname :
        f.writerow([yname])
    f.writerows(zip(y_pred))
    out.close()


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

def main():
    # load Titanic dataset
    titanic = load_data("titanic_train.csv", header=1, predict_col=0)
    X = titanic.X; Xnames = titanic.Xnames
    y = titanic.y; yname = titanic.yname
    n,d = X.shape  # n = number of examples, d =  number of features



    #========================================
    # train Majority Vote classifier on data
    print 'Classifying using Majority Vote...'
    clf = MajorityVoteClassifier() # create MajorityVote classifier, which includes all model parameters
    clf.fit(X, y)                  # fit training data using the classifier
    y_pred = clf.predict(X)        # take the classifier and run it on the training data
    train_error = 1 - metrics.accuracy_score(y, y_pred, normalize=True)
    print '\t-- training error: %.3f' % train_error



    ### ========== TODO : START ========== ###
    # part a: evaluate training error of Decision Tree classifier
    print 'Classifying using Decision Tree...'
    clf = DecisionTreeClassifier(criterion="entropy")
    clf.fit(X, y)
    y_pred = clf.predict(X)
    train_error = 1 - metrics.accuracy_score(y, y_pred, normalize=True)
    print '\t-- training error: %.3f' % train_error

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



    # note: uncomment out the following lines to output the Decision Tree graph
    """
    # save the classifier -- requires GraphViz and pydot
    import StringIO, pydot
    from sklearn import tree
    dot_data = StringIO.StringIO()
    tree.export_graphviz(clf, out_file=dot_data,
                         feature_names=Xnames,
                         class_names=["Died", "Survived"])
    graph = pydot.graph_from_dot_data(dot_data.getvalue())
    graph.write_pdf("dtree.pdf")
    """
```

```python



    ### ========== TODO : START ========== ###
    # part b: use cross-validation to compute average training and test error of classifiers
    print 'Investigating various classifiers...'

    print 'MajorityVoteClassifier Error'
    train_error, test_error = error(MajorityVoteClassifier() , X, y, ntrials=100, test_size=0.2)
    print '\t-- training error: %.3f' % train_error
    print '\t-- test error: %.3f' % test_error

    print 'RandomClassifier Error'
    train_error, test_error = error(RandomClassifier() , X, y, ntrials=100, test_size=0.2)
    print '\t-- training error: %.3f' % train_error
    print '\t-- test error: %.3f' % test_error

    print 'DecisionTreeClassifier Error'
    train_error, test_error = error(DecisionTreeClassifier(criterion="entropy"), X, y, ntrials=100, test_size=0.2)
    print '\t-- training error: %.3f' % train_error
    print '\t-- test error: %.3f' % test_error



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



    ### ========== TODO : START ========== ###
    # part c: investigate decision tree classifier with various depths
    print 'Investigating depths...'
    clfs = [MajorityVoteClassifier(), RandomClassifier()]
    label_clfs = ["MVC", "RC"]
    for clf_index in range(0,len(clfs)):
        train_error_arr = []
        test_error_arr =[]
        depths = range(1,21)
        for depth in depths:
            clf = clfs[clf_index]
            train_error, test_error = error(clf,X,y)
            train_error_arr.append(train_error)
            test_error_arr.append(test_error)

        plt.plot(depths, train_error_arr, label = label_clfs[clf_index] + " train error")
        plt.plot(depths, test_error_arr, label = label_clfs[clf_index] + " test error")

    train_error_arr = []
    test_error_arr =[]
    # only do the DecisionTreeClassifier One
    for depth in depths:
        clf = DecisionTreeClassifier(criterion="entropy", max_depth=depth)
        train_error, test_error = error(clf,X,y)
        train_error_arr.append(train_error)
        test_error_arr.append(test_error)

    plt.plot(depths, train_error_arr, label = "DT train error")
    plt.plot(depths, test_error_arr, label = "DT test error")
    plt.legend()
    plt.xlabel("Max Depth")
    plt.ylabel("Average Error")
    plt.show()
    ### ========== TODO : END ========== ###




    ### ========== TODO : START ========== ###
    # part d: investigate decision tree classifier with various training set sizes
    print 'Investigating training set sizes...'

    clfs = [MajorityVoteClassifier(), RandomClassifier(), DecisionTreeClassifier(criterion="entropy", max_depth=6)]
    label_clfs = ["MVC", "RC", "DT"]
    split_sizes =  map (lambda u: u*.05,  range(1,20))
    training_splits = map (lambda u: 1 - u,  split_sizes)

    for clf_index in range(0, len(clfs)):
        train_error_arr = []
        test_error_arr =[]
        for split in split_sizes:
            clf = clfs[clf_index]
            train_error, test_error = error(clf,X,y,test_size=split)
            train_error_arr.append(train_error)
            test_error_arr.append(test_error)

        plt.plot(training_splits, train_error_arr, label = label_clfs[clf_index] + " train error")
        plt.plot(training_splits, test_error_arr, label = label_clfs[clf_index] +" test error")

    plt.xlabel("Training Data Size")
    plt.ylabel("Average Error")
    plt.legend()
    plt.show()


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



    ### ========== TODO : START ========== ###
    # Contest
    # uncomment write_predictions and change the filename

    # evaluate on test data
    titanic_test = load_data("titanic_test.csv", header=1, predict_col=None)
    X_test = titanic_test.X
    y_pred = clf.predict(X_test)   # take the trained classifier and run it on the test data
    #write_predictions(y_pred, "../data/yjw_titanic.csv", titanic.yname)

    ### ========== TODO : END ========== ###
    print 'Done'
```

```python

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