```python
"""
Author      : Yi-Chieh Wu, Sriram Sankararaman
Description : Titanic
"""

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


from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score
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
        """
        majority_val = Counter(y).most_common(1)[0][0]
        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 b: set self.probabilities_ according to the training set
        random_val = Counter(y)
        self.probabilities_ = [random_val[0]/float(y.shape[0]) , random_val[1]/float(y.shape[0])]

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

        y = np.random.choice([0,1], size=X.shape[0], replace=True, p=self.probabilities_)

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

        return y


######################################################################
# functions
######################################################################
def plot_histograms(X, y, Xnames, yname) :
    n,d = X.shape  # n = number of examples, d =  number of features
    fig = plt.figure(figsize=(20,15))
    nrow = 3; ncol = 3
    for i in range(d) :
        fig.add_subplot (3,3,i+1)
        data, bins, align, labels = plot_histogram(X[:,i], y, Xname=Xnames[i], yname=yname, show = False)
        n, bins, patches = plt.hist(data, bins=bins, align=align, alpha=0.5, label=labels)
        plt.xlabel(Xnames[i])
        plt.ylabel('Frequency')
        plt.legend() #plt.legend(loc='upper left')

    plt.savefig ('histograms.pdf')



def plot_histogram(X, y, Xname, yname, show = True) :
    """
    Plots histogram of values in X grouped by y.

    Parameters
    --------------------
        X     -- numpy array of shape (n,d), feature values
        y     -- numpy array of shape (n,), target classes
        Xname -- string, name of feature
        yname -- string, name of target
    """

    # set up data for plotting
    targets = sorted(set(y))
    data = []; labels = []
    for target in targets :
        features = [X[i] for i in range(len(y)) if y[i] == target]
        data.append(features)
        labels.append('%s = %s' % (yname, target))

    # set up histogram bins
    features = set(X)
    nfeatures = len(features)
    test_range = list(range(int(math.floor(min(features))), int(math.ceil(max(features)))+1))
    if nfeatures < 10 and sorted(features) == test_range:
        bins = test_range + [test_range[-1] + 1] # add last bin
        align = 'left'
    else :
        bins = 10
        align = 'mid'

    # plot
    if show == True:
```

```
        plt.figure()
        n, bins, patches = plt.hist(data, bins=bins, align=align, alpha=0.5, label=labels)
        plt.xlabel(Xname)
        plt.ylabel('Frequency')
        plt.legend() #plt.legend(loc='upper left')
        plt.show()

    return data, bins, align, labels


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

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

    ### ========== TODO : START ========== ###
    # compute cross-validation error over ntrials
    # hint: use train_test_split (be careful of the parameters)

    train_error = 0
    test_error = 0

    for i in range(0, ntrials):
        X_train, X_test, y_train, y_test = train_test_split (X, y, test_size= test_size, random_state=i )
        clf.fit(X_train, y_train)
        y_train_pred = clf.predict(X_train)
        train_error += 1-metrics.accuracy_score(y_train, y_train_pred, normalize=True)
        y_test_pred = clf.predict(X_test)
        test_error += 1-metrics.accuracy_score(y_test, y_test_pred, normalize=True)

    train_error = train_error/ntrials
    test_error = test_error/ntrials

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

    return train_error, test_error

def error_h(clf, X, y, ntrials=100, test_size=0.2, inc=10) :
    """
    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

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

```python
    ### ========== TODO : START ========== ###
    # compute cross-validation error over ntrials
    # hint: use train_test_split (be careful of the parameters)

    train_error = 0
    test_error = 0
    inc = inc/10.0

    for i in range(0, ntrials):
        X_train, X_test, y_train, y_test = train_test_split (X, y, test_size= test_size, random_state=i )
        X_train = X_train[:int(len(X_train)*inc)]
        y_train = y_train[:int(len(y_train)*inc)]
        clf.fit(X_train, y_train)
        y_train_pred = clf.predict(X_train)
        train_error += 1-metrics.accuracy_score(y_train, y_train_pred, normalize=True)
        y_test_pred = clf.predict(X_test)
        test_error += 1-metrics.accuracy_score(y_test, y_test_pred, normalize=True)

    train_error = train_error/ntrials
    test_error = test_error/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(list(zip(y_pred)))
    out.close()


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

def main():
    # load Titanic dataset
    titanic = load_data("../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



    #========================================
    # part a: plot histograms of each feature
    print('Plotting...')
    plot_histograms(X, y, Xnames=Xnames, yname=yname)


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

```python
### ========== TODO : START ========== ###
# part b: evaluate training error of Random classifier
print('Classifying using Random...')
rclf = RandomClassifier() # create MajorityVote classifier, which includes all model parameters
rclf.fit(X, y)                  # fit training data using the classifier
ry_pred = rclf.predict(X)        # take the classifier and run it on the training data
train_error = 1 - metrics.accuracy_score(y, ry_pred, normalize=True)
print('\t-- training error: %.3f' % train_error)

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



### ========== TODO : START ========== ###
# part c: evaluate training error of Decision Tree classifier
# use criterion of "entropy" for Information gain
print('Classifying using Decision Tree...')
dclf = DecisionTreeClassifier(criterion="entropy") # create MajorityVote classifier, which includes all model parameters
dclf.fit(X, y)                  # fit training data using the classifier
dy_pred = dclf.predict(X)        # take the classifier and run it on the training data
train_error = 1 - metrics.accuracy_score(y, dy_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)
graph = pydot.graph_from_dot_data(dot_data.getvalue())
graph.write_pdf("dtree.pdf")
"""



### ========== TODO : START ========== ###
# part d: evaluate training error of k-Nearest Neighbors classifier
# use k = 3, 5, 7 for n_neighbors
print('Classifying using k-Nearest Neighbors...')
kclf = KNeighborsClassifier(n_neighbors=3) # create MajorityVote classifier, which includes all model parameters
kclf.fit(X, y)                  # fit training data using the classifier
ky_pred = kclf.predict(X)        # take the classifier and run it on the training data
train_error = 1 - metrics.accuracy_score(y, ky_pred, normalize=True)
print('\t-- training error for k=3: %.3f' % train_error)

k5clf = KNeighborsClassifier(n_neighbors=5) # create MajorityVote classifier, which includes all model parameters
k5clf.fit(X, y)                  # fit training data using the classifier
k5y_pred = k5clf.predict(X)        # take the classifier and run it on the training data
train_error = 1 - metrics.accuracy_score(y, k5y_pred, normalize=True)
print('\t-- training error for k=5: %.3f' % train_error)

kclf = KNeighborsClassifier(n_neighbors=7) # create MajorityVote classifier, which includes all model parameters
kclf.fit(X, y)                  # fit training data using the classifier
ky_pred = kclf.predict(X)        # take the classifier and run it on the training data
train_error = 1 - metrics.accuracy_score(y, ky_pred, normalize=True)
print('\t-- training error for k=7: %.3f' % train_error)

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

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

mclf_error = error(mclf, X, y)
print('Majority Vote for training and test error...')
print('\t-- average training error: %.3f' % mclf_error[0])
print('\t-- average testing error: %.3f' % mclf_error[1])

rclf_error = error(rclf, X, y)
print('Random for training and test error...')
print('\t-- average training error: %.3f' % rclf_error[0])
print('\t-- average testing error: %.3f' % rclf_error[1])

dclf_error = error(dclf, X, y)
print('Decision Tree for training and test error...')
print('\t-- average training error: %.3f' % dclf_error[0])
print('\t-- average testing error: %.3f' % dclf_error[1])

k5clf_error = error(k5clf, X, y)
print('k-Nearest Neighbors (k=5) for training and test error...')
print('\t-- average training error: %.3f' % k5clf_error[0])
print('\t-- average testing error: %.3f' % k5clf_error[1])



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



### ========== TODO : START ========== ###
# part f: use 10-fold cross-validation to find the best value of k for k-Nearest Neighbors classifier
print('Finding the best k for KNeighbors classifier...')
plt.clf()
k_val = []
scores_err = []

for i in range(1, 51, 2):
    knclf = KNeighborsClassifier(n_neighbors=i)
    scores = cross_val_score (knclf, X, y, cv=10)
    print('\t-- cross val score for k=%d: %.3f' % (i, 1-np.mean(scores)))
    scores_err.append(1-np.mean(scores))
    k_val.append(i)

plt.xlabel("k")
plt.ylabel('cross validation error value')
plt.plot(k_val, scores_err)

plt.savefig ('kneighbors_cross_val.pdf')




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



### ========== TODO : START ========== ###
# part g: investigate decision tree classifier with various depths
print('Investigating depths...')
plt.clf()
d_val = []
train_err = []
test_err = []
```

```
for i in range(1,21):
    ddclf = DecisionTreeClassifier(criterion='entropy', max_depth=i)
    train_error, test_error = error(ddclf, X, y)

    train_err.append(train_error)
    test_err.append(test_error)
    print('\t-- cross val score for k=%d: %.3f' % (i, np.mean(test_err)))
    d_val.append(i)

plt.xlabel("depths")
plt.ylabel('cross validation error value')
plt.plot(d_val, train_err, label ='training error')
plt.plot(d_val, test_err, label ='test error')
plt.legend()

plt.savefig ('decisiontree_cross_val.pdf')



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



### ========== TODO : START ========== ###
# part h: investigate Decision Tree and k-Nearest Neighbors classifier with various training set sizes
print('Investigating training set sizes...')
plt.clf()
hknclf = KNeighborsClassifier(n_neighbors=7)
hdclf = DecisionTreeClassifier(max_depth=7)

hd_training_error = []
hd_test_error = []
hkn_training_error = []
hkn_test_error = []

x_vals = []
for i in range(1, 11, 1):
    hd_err = error_h(hdclf, X, y, 100, 0.1, i)
    hkn_err = error_h(hknclf, X, y, 100, 0.1, i)

    hd_training_error.append(hd_err[0])
    hd_test_error.append(hd_err[1])
    hkn_training_error.append(hkn_err[0])
    hkn_test_error.append(hkn_err[1])
    x_vals.append(i*10)


    print('    Using %g percent of training data...' % (i*10))
    print('\t decision tree...')
    print ('\t\t training error: %.3f  and testing error: %.3f' % (hd_err[0], hd_err[1]))
    print('\t k neighbors...')
    print ('\t\t training error: %.3f and testing error: %.3f' % (hkn_err[0], hkn_err[1]))

plt.xlabel ("Training set percentage used")
plt.ylabel('Error')
plt.plot(x_vals, hd_training_error, label='Decision Tree Training Error' )
plt.plot(x_vals, hd_test_error, label='Decision Tree Test Error' )

plt.plot(x_vals, hkn_training_error, label='KNN Training Error' )
plt.plot(x_vals, hkn_test_error, label='KNN Test Error' )
plt.legend()

plt.savefig("part_h.pdf")
```

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


    print('Done')


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