```
#!/usr/bin/env python
import numpy as np
from math import floor
from numpy import linalg as la
from numpy import matlib as matlib
import matplotlib.pyplot as plt
import argparse
import os
import pdb
from scipy import spatial
import time
import operator
import random
import warnings

FISHER = 0
RANDOM_CLASSIFIER = 0
def binaryClassify_fisher(train_features, train_truth, test_features, test_truth, classa, classb,
max_iterations):
    print 5000-max_iterations
    if max_iterations <= 0:
        print "Something went wrong, the maximum recursion depth was exceeded"
        quit()
    max_iterations -= 1
    #perform a classification on the data
    if FISHER or RANDOM_CLASSIFIER:
      train_classification_result, test_classification_result = FisherClassifier(train_features,
train_truth, test_features, classa, classb)
    else: #linear regression
      train_classification_result, test_classification_result = LinearRegression(train_features,
train_truth, test_features, classa, classb)
    #check to see if either classification is "pure"
    #select the items which are really in class 0
    class_a_samples = train_classification_result[train_truth == classa]
    class_b_samples = train_classification_result[train_truth == classb]
    print "class_a_train_rate %f"%(float(np.sum(class_a_samples))/class_a_samples.shape[0])
    print "class_b_train_rate %f"%(1.0 - float(np.sum(class_b_samples))/class_b_samples.shape[0])
    class_a_test_samples = test_classification_result[test_truth == classa]
    class_b_test_samples = test_classification_result[test_truth == classb]
    print "class_a_test_rate %f"%(float(np.sum(class_a_test_samples))/class_a_test_samples.shape[0])
    print "class_b_test_rate %f"%(1.0 - float(np.sum(class_b_test_samples))/class_b_test_samples.shape[0])
    #sum errors from every recursion
    num_a_errors = 0
    num_b_errors = 0
    if not np.all(class_a_samples == classa) and not np.all(class_a_samples == classb) and not np.all
(test_truth[test_classification_result == classa]) and np.any(test_truth[test_classification_result ==
classa]) and not np.all(train_truth[train_classification_result == classa]) and np.any(train_truth
[train_classification_result == classa]):
        #recurse on this branch
  new_train_features = train_features[train_classification_result==classa]
        new_train_truth = train_truth[train_classification_result == classa]
        new_test_features = test_features[test_classification_result==classa]
        new_test_truth = test_truth[test_classification_result == classa]
        print "left test: %4d, left train: %4d"%(new_test_truth.shape[0],new_train_truth.shape[0])
        tempa, tempb = binaryClassify_fisher(new_train_features, new_train_truth, new_test_features,
new_test_truth, classa, classb, max_iterations)
        num_a_errors += tempa
        num_b_errors += tempb
    else:
        #the sample was "pure" so result results on it
        #compute the number of errors in this dataset
        #check what the previous output was (we only have to check the first element since they are all
the same
        num_a_errors += test_truth[test_truth==classa].shape[0] - np.sum(test_classification_result
[test_truth == classa] == classa)
    if not np.all(class_b_samples == classb) and not np.all(class_b_samples == classa) and not np.all
(test_truth[test_classification_result == classb]) and np.any(test_truth[test_classification_result ==
```

```
classb]) and not np.all(train_truth[train_classification_result == classb]) and np.any(train_truth
[train_classification_result == classb]):
        #recurse on this branch
  new_train_features = train_features[train_classification_result==classb]
        new_train_truth = train_truth[train_classification_result == classb]
        new_test_features = test_features[test_classification_result==classb]
        new_test_truth = test_truth[test_classification_result == classb]
        print "right test: %4d, right train: %4d"%(new_test_truth.shape[0],new_train_truth.shape[0])
        tempa,tempb = binaryClassify_fisher(new_train_features, new_train_truth, new_test_features,
new_test_truth, classa, classb, max_iterations)
        num_a_errors += tempa
        num_b_errors += tempb
    else:
        #the sample was "pure" so result results on it
        #compute the number of errors in this dataset
        #check what the previous output was (we only have to check the first element since they are all
the same
        num_b_errors += test_truth[test_truth == classb].shape[0] - np.sum(test_classification_result
[test_truth == classb] == classb)

    return num_a_errors, num_b_errors

def LinearRegression(train_features, train_truth, test_features, classa, classb):
     train_truth_internal = np.matrix(train_truth.copy()).T
     #make classification 0-centered
     train_truth_internal[train_truth_internal == 0] = -1
     filter = la.inv(train_features.T * train_features + np.eye(train_features.shape[1])*1e-10)  *
train_features.T * train_truth_internal
     test_classification = (test_features * filter)
     train_classification = (train_features * filter)
     test_rtn = test_classification.copy()
     train_rtn = train_classification.copy()
     test_rtn[test_classification <= 0] = classa
     train_rtn[train_classification <= 0] = classa
     test_rtn[test_classification >0] = classb
     train_rtn[train_classification >0] = classb
     return np.array(train_rtn)[:,0],np.array(test_rtn)[:,0]

def FisherClassifier(train_features, train_truth, test_features, classa, classb):
  with warnings.catch_warnings():
    warnings.filterwarnings('error')
    '''
    :param features:
    :param classification:
    :param test_data:
    :return:
    '''
    # separate classes
    class_a_features = train_features[train_truth == classa]
    class_b_features = train_features[train_truth == classb]
    try:
      class_a_mean = np.mean(class_a_features, 0).T
      class_a_cov  = np.cov(class_a_features.T)

      class_b_mean = np.mean(class_b_features, 0).T
      class_b_cov  = np.cov(class_b_features.T)
    except Warning:
      #there was no covariance computed, so just assign everything to one class
      if class_b_features.shape[0] < 2:
        #send eveything to class a
        return [np.ones(train_features.shape[0])*classa, np.ones(test_features.shape[0])*classa]
      else:
        return [np.ones(train_features.shape[0])*classb, np.ones(test_features.shape[0])*classb]
    # compute the Fisher criteria projection to one dimension
    element_classified_a = False
    element_classified_b = False
    while not element_classified_a or not element_classified_b:
```

```
      if FISHER:
        try:
          projection = la.inv(class_a_cov + class_b_cov + np.eye(class_a_cov.shape[0])*10e-15) *
(class_a_mean - class_b_mean)
        except:
          pdb.set_trace()
      else: #if RANDOM_CLASSIFIER
        projection = np.matrix(np.zeros(class_a_cov.shape[0]))
        for idx in range(projection.shape[0]):
          projection[idx] = random.random()
        projection = projection.T
      projection = projection / la.norm(projection)
      element_classified_a = False
      element_classified_b = False
      # project all of the data
      class_a_projection = class_a_features * projection
      class_b_projection = class_b_features * projection

      class_a_gauss_build = GaussianBuild(class_a_projection)
      class_b_gauss_build = GaussianBuild(class_b_projection)

      #classify the test data
      test_classification_result = []
      for sample in test_features:
          try:
              sample_projection = sample * projection
          except ValueError:
              pdb.set_trace()
          class_a_prob = ComputeGaussianProbability(class_a_gauss_build[0], class_a_gauss_build[1],
sample_projection)
          class_b_prob = ComputeGaussianProbability(class_b_gauss_build[0], class_b_gauss_build[1],
sample_projection)
          if class_a_prob > class_b_prob:
              test_classification_result.append(classa)
          else:
              test_classification_result.append(classb)
      #classify the train data
      train_classification_result = []
      for sample in train_features:
          try:
              sample_projection = sample * projection
          except ValueError:
              pdb.set_trace()
          class_a_prob = ComputeGaussianProbability(class_a_gauss_build[0], class_a_gauss_build[1],
sample_projection)
          class_b_prob = ComputeGaussianProbability(class_b_gauss_build[0], class_b_gauss_build[1],
sample_projection)
          if class_a_prob > class_b_prob:
              train_classification_result.append(classa)
              element_classified_a = True
          else:
              train_classification_result.append(classb)
              element_classified_b = True
      if FISHER:
        break
    return [np.array(train_classification_result).T,np.array(test_classification_result).T]

def GaussianBuild(features):
    """
    computes the mean and covariance for a dataset
    :param features: s x f np.matrix (s samples by f features)
    :param classification: s x 1 np.ndarray
    :param class_id: scalar value to find
    :return: [covariance(f x f),mean (f x 1)]
    """
    #print 'Of ', features.shape, 'Elements, ', features.shape
    cov_mat = np.cov(features.T)
```

```
    mean_mat = np.mean(features.T)
    return [cov_mat, mean_mat]


def ComputeGaussianProbability(cov_mat, mean_mat, sample):
    """
    computes the probability of a particular sample belonging to a particular gaussian distribution
    :param cov_mat: f x f np.matrix (f features)
    :param mean_mat: f x 1 np.matrix
    :param sample: f x 1 np.matrix
    :return:
    """
    mean_mat = np.matrix(mean_mat).T
    sample = sample.T
    # sample = meanMat
    non_invertible = True
    eye_scale = 0.0
    try:
      cov_mat_inverse = 1.0 / cov_mat
    except Warning:
      cov_mat_inverse = 1
      cov_mat = 1
    probability = 1.0 / (np.sqrt(la.norm(2 * np.pi * cov_mat)))
    probability *= np.exp(-0.5 * (sample - mean_mat).T * cov_mat_inverse * (sample - mean_mat))
    return probability

def ParseData(raw_data):
    raw_data = raw_data.rstrip('\n')
    raw_data_list = raw_data.split('\n')
    data_list = list()
    for raw_data_point in raw_data_list:
        raw_data_point = raw_data_point.rstrip()
        point = raw_data_point.split(' ')
        data_list.append([float(x) for x in point])
    data_list.pop()
    data_list_np = np.array(data_list)
    return data_list_np

def main():
    parser = argparse.ArgumentParser(description='Process input')
    parser.add_argument('-t', '--training_file', type=str, help='submit data to train against')
    parser.add_argument('-d', '--traintest_file', type=str, help='List indicating which data is training 
vs. test')

    args = parser.parse_args()
    print os.getcwd()
    # Check if Arguments allow execution
    if (not args.training_file):
        print "Error: No training Data or model present!"
        return -1
    with open(args.traintest_file) as file:
        raw_data = file.read()
        traintest_data = np.array(ParseData(raw_data)[:,0])

    if args.training_file:
        # trainagainst training file
        if not os.path.isfile(args.training_file):
            print "Error: Training file doesn't exist!"
            return -1
        # train
        with open(args.training_file) as file:
            # read file contents
            raw_data = file.read()
            # parse data
            train_data = ParseData(raw_data)
            test_truth = train_data[traintest_data==1,-1]
            test_features = np.matrix(np.array(train_data[traintest_data==1,0:-1]))
```

```
            train_truth = train_data[traintest_data==0,-1]
            train_features = np.matrix(np.array(train_data[traintest_data==0,0:-1]))
            #make the data homogeneous
            homogeneous_col = np.ones([train_features.shape[0],1]);
            train_features = np.append(train_features, homogeneous_col,axis=1)
            homogeneous_col = np.ones([test_features.shape[0],1]);
            test_features = np.append(test_features, homogeneous_col, axis=1)
    try:
        test_features
        train_features
    except NameError:
        print "You must provide test and training data"
        quit()
    #iteratively call the classifier to build a binary tree until the classification is perfect or a
timeout is reached
    max_iterations = 5000
    #sort the data into test and training sets
    with open(args.traintest_file) as file:
        raw_data = file.read()
        traintest_data = ParseData(raw_data)
    num_a_errors,num_b_errors = binaryClassify_fisher(train_features, train_truth, test_features,
test_truth, 0, 1, max_iterations)
    classa = 0
    classb = 1
    print "Total a errors: %d of %d, %% error:%f"%(num_a_errors,test_truth[test_truth==classa].shape
[0],float(num_a_errors)/test_truth[test_truth==classa].shape[0])
    print "Total b errors: %d of %d, %% error:%f"%(num_b_errors,test_truth[test_truth==classb].shape
[0],float(num_b_errors)/test_truth[test_truth==classb].shape[0])
if __name__ == '__main__':
    main()
```