Python 2

from \_\_future\_\_ import division

# import matplotlib.pyplot as plt

# %matplotlib inline

import numpy as np

import pandas as pd

from sklearn.ensemble import GradientBoostingClassifier as GBC

from sklearn.ensemble import RandomForestClassifier as RF

# We will have the latest version on AWS scikit-learn-0.18.1

# from sklearn.grid\_search import GridSearchCV  # sklearn 0.17

from sklearn.model\_selection import train\_test\_split, ParameterGrid

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score

from sklearn.metrics import f1\_score, confusion\_matrix, roc\_curve, auc

import pickle

import os

def get\_data(path):

    '''

    loads the data specified on the path and puts it on a dataframe

    input: path (srt)

    output: dataframe

    '''

    x = np.load(path)['x']

    # changed "classification" by "is\_fishing"

    x = x[~np.isinf(x['is\_fishing']) & ~np.isnan(x['is\_fishing']) & ~np.isnan(x['timestamp']) & ~np.isnan(x['speed']) & ~np.isnan(x['course'])]

    df = pd.DataFrame(x)

    return df

def get\_all\_data(dir = None):

    '''

    it will loop over the files in dir and load them

    input: directory name (srt)

    output: dictionary of dataframes with key = name of file

    '''

    if dir is None:

        dir = os.path.join(os.path.dirname(\_\_file\_\_), '.', 'data/labeled')

    datasets = {}

    for filename in os.listdir(dir):

        if filename.endswith('.measures.labels.npz'):

            name = filename[:-len('.measures.labels.npz')]

            #             datasets[name] = dict(zip(['all', 'train', 'cross', 'test'], load\_dataset\_by\_vessel(os.path.join(dir, filename))))

            x = np.load(os.path.join(dir, filename))['x']

            x = x[~np.isinf(x['is\_fishing']) & ~np.isnan(x['is\_fishing']) & ~np.isnan(x['timestamp']) & ~np.isnan(x['speed']) & ~np.isnan(x['course'])]

            datasets[name] = pd.DataFrame(x)

            # print name

    return datasets

def get\_group(data\_dict,gear\_type):

    # data\_dict = get\_all\_data('data/labeled')

    group\_per\_gear = {

    'longliners':

    ['alex\_crowd\_sourced\_Drifting\_longlines',

    'kristina\_longliner\_Drifting\_longlines',

    'pybossa\_project\_3\_Drifting\_longlines'],

    'purse\_seines':

    ['alex\_crowd\_sourced\_Purse\_seines',

    'kristina\_ps\_Purse\_seines',

    'pybossa\_project\_3\_Purse\_seines'],

    'trawlers':

    ['kristina\_trawl\_Trawlers',

    'pybossa\_project\_3\_Trawlers'],

    'others':

    ['alex\_crowd\_sourced\_Unknown',

    'kristina\_longliner\_Fixed\_gear',

    'kristina\_longliner\_Unknown',

    'kristina\_ps\_Unknown',

    'pybossa\_project\_3\_Unknown',

    'pybossa\_project\_3\_Pole\_and\_line',

    'pybossa\_project\_3\_Trollers'

    'pybossa\_project\_3\_Fixed\_gear'],

    'false\_positives':

    ['false\_positives\_Drifting\_longlines',

    'false\_positives\_Fixed\_gear',

    'false\_positives\_Purse\_seines',

    'false\_positives\_Trawlers',

    'false\_positives\_Unknown']

    }

    df = pd.concat([ data\_dict[filename] for filename in group\_per\_gear[gear\_type]])

    df = df.reset\_index()

    df = df.drop('index',axis=1)

    return df

# def train\_model(model,X\_train,y\_train,model\_name):

#   model.fit(X\_train, y\_train)

#   return model #, probabilities

def pickle\_model(model,model\_name):

    file\_name = 'results/{}.pkl'.format(model\_name)

    with open(file\_name, 'w') as f:

        pickle.dump(model, f)

    pass

def roc\_curve(probabilities, labels):

    '''

    Sort instances by their prediction strength (the probabilities)

    For every instance in increasing order of probability:

        Set the threshold to be the probability

        Set everything above the threshold to the positive class

        Calculate the True Positive Rate (aka sensitivity or recall)

        Calculate the False Positive Rate (1 - specificity)

    Return three lists: TPRs, FPRs, thresholds

    '''

    prob\_order = np.sort(probabilities)

    TPRs = []

    FPRs = []

    for pr in prob\_order:

        th = pr

        classification = probabilities >= th  # boolean

        TP = sum(np.logical\_and(classification,labels))

        TPRs.append(float(TP))

        labels\_neg = -1\*labels + 1

        FP = sum(np.logical\_and(classification, labels\_neg))

        FPRs.append(float(FP))

    P = sum(labels)

    N = len(labels) - P

    return TPRs/P, FPRs/N, prob\_order

def do\_grid\_search(est,param\_grid,X\_train,y\_train,X\_cross,y\_cross):

    # Initalize our model here

    # est = GBC()

    # param\_grid\_GBC = {'learning\_rate': [0.05, 0.1, 0.2],

    #           'max\_depth': [2, 5, 10],

    #           'min\_samples\_leaf': [3, 10, 20]

    #           }

    # sklearn.model\_selection.ParameterGrid(param\_grid)

    # predict\_proba(X)

    best\_score = 0.5

    for g in ParameterGrid(param\_grid):

        print ('running: ', str(g)  )# more verbose

        est.set\_params(\*\*g)

        est.fit(X\_train,y\_train)

        cross\_score = est.score(X\_cross,y\_cross) # mean accuracy

        train\_score = est.score(X\_train,y\_train) # mean accuracy

        if cross\_score > best\_score:

            best\_cross\_score = cross\_score

            best\_train\_score = train\_score

            best\_params = g

    # gs\_cv = GridSearchCV(est, param\_grid, cv=2, n\_jobs=-1, verbose=2).fit(X\_train, y\_train)

    # return gs\_cv.best\_score\_, gs\_cv.best\_params\_

    return best\_train\_score, best\_cross\_score, best\_params

def get\_scores(fitted\_classifier, X\_test, y\_test):

    # model = classifier(\*\*kwargs)

    # model.fit(X\_train, y\_train)

    model = fitted\_classifier

    y\_predict = model.predict(X\_test)

    C = confusion\_matrix(y\_test, y\_predict)

    # confusion\_matrix(y\_true, y\_pred)

    # true negatives is C\_{0,0},

    # false negatives is C\_{1,0},

    # true positives is C\_{1,1},

    # false positives is C\_{0,1}.

    # P = C\_{1,1} + C\_{1,0}

    accurary = (C[0][0]+C[1][1]) / len(y\_test)

    precision = C[1][1] / (C[0][1] + C[1][1])

    recall = C[1][1] / (C[1][1] + C[1][0])

    F1 = 2 \* (precision \* recall) / (precision + recall)

    # Accuracy, Recall, F1-score

    # return model.score(X\_test, y\_test), recall, F1

    return accurary, recall, F1

# ==== From here on, modified from SkyTruth's repo =========

def is\_fishy(x):

    return x['is\_fishing'] == 1

def fishy(x):

    return x[is\_fishy(x)]

def nonfishy(x):

    return x[~is\_fishy(x)]

def \_subsample\_even(x0, mmsi, n):

    """Return `n` subsamples from `x0`

    - all samples have given `mmsi`

    - samples are evenly divided between fishing and nonfishing

    """

    # Create a mask that is true whenever mmsi is one of the mmsi

    # passed in

    mask = np.zeros([len(x0)], dtype=bool)

    for m in mmsi:

        mask |= (x0['mmsi'] == m)

    x = x0[mask]  # this makes is a np array?? nope...

    # Pick half the values from fishy rows and half from nonfishy rows.

    f = fishy(x)

    nf = nonfishy(x)

    if n//2 > len(f) or n//2 > len(nf):

        warnings.warn("insufficient items to sample, returning fewer")

    f\_index = np.random.choice(f.index, min(n//2, len(f)), replace=False)

    nf\_index = np.random.choice(nf.index, min(n//2, len(nf)), replace=False)

    f = f.ix[f\_index]

    nf = nf.ix[nf\_index]

    # nf = np.random.choice(nf, min(n//2, len(nf)), replace=False)

    ss = pd.concat([f, nf])  #this was making it a np array! yes

    # np.random.shuffle(ss) # no shuffling

    return ss

def \_subsample\_proportional(x0, mmsi, n):

    """Return `n` subsamples from `x0`

    - all samples have given `mmsi`

    - samples are random, so should have ~same be in the same proportions

      as the x0 for the given mmsi.

    """

    # Create a mask that is true whenever mmsi is one of the mmsi

    # passed in

    mask = np.zeros([len(x0)], dtype=bool)

    for m in mmsi:

        mask |= (x0['mmsi'] == m)

    x = x0[mask]

    # Pick values randomly

    # Pick values randomly

    # ====DEBUGGER=======

    # import pdb

    # pdb.set\_trace()

    if n > len(x):

        warnings.warn("Warning, inufficient items to sample, returning {}".format(len(x)))

        n = len(x)

    x\_index = np.random.choice(x.index, n, replace=False)

    # ss = np.random.choice(x, n, replace=False)

    ss = x.ix[x\_index]

    # np.random.shuffle(ss) # the shuffeling is giving me trouble

    return ss

def sample\_by\_vessel(x, size = 20000, even\_split=None, seed=4321):

    # def load\_dataset\_by\_vessel(path, size = 20000, even\_split=None, seed=4321):

    """Load a dataset from `path` and return train, valid and test sets

    path - path to the dataset

    size - number of samples to return in total, divided between the

           three sets as (size//2, size//4, size//4)

    even\_split - if True, use 50/50 fishing/nonfishing split for training

                  data, otherwise sample the data randomly.

    The data at path is first randomly divided by divided into

    training (1/2), validation (1/4) and test(1/4) data sets.

    These sets are chosen so that MMSI values are not shared

    across the datasets.

    The validation and test data are sampled randomly to get the

    requisite number of points. The training set is sampled randomly

    if `even\_split` is False, otherwise it is chose so that half the

    points are fishing.

    """

    # Set the seed so that we can reproduce results consistently

    np.random.seed(seed)

    # # Load the dataset and strip out any points that aren't classified

    # # (has'is\_fishing == Inf)

    # x = np.load(path)['x']

    # x = x[~np.isinf(x['is\_fishing']) & ~np.isnan(x['is\_fishing']) & ~np.isnan(x['timestamp']) & ~np.isnan(x['speed']) & ~np.isnan(x['course'])]

    if size > len(x):

        print ("Warning, insufficient items to sample, returning all")

        size = len(x)

    # Get the list of MMSI and shuffle them. The compute the cumulative

    # lengths so that we can divide the points ~ evenly. Use search

    # sorted to find the division points

    mmsi = list(set(x['mmsi']))

    if even\_split is None:

        even\_split = x['is\_fishing'].sum() > 1 and x['is\_fishing'].sum() < len(x)

    if even\_split:

        base\_mmsi = mmsi

        # Exclude mmsi that don't have at least one fishing or nonfishing point

        mmsi = []

        for m in base\_mmsi:

            subset = x[x['mmsi'] == m]

            fishing\_count = subset['is\_fishing'].sum()

            if fishing\_count == 0 or fishing\_count == len(subset):

                continue

            mmsi.append(m)

    np.random.shuffle(mmsi)

    nx = len(x)

    sums = np.cumsum([(x['mmsi'] == m).sum() for m in mmsi])

    n1, n2 = np.searchsorted(sums, [nx//2, 3\*nx//4])

    if n2 == n1:

        n2 += 1

    # ====DEBUGGER=======

    # import pdb

    # pdb.set\_trace()

    train\_subsample = \_subsample\_even if even\_split else \_subsample\_proportional

# try:

    xtrain = train\_subsample(x, mmsi[:n1], size//2)

    # xtrain = \_subsample\_proportional(x, mmsi[:n1], size//2)

    xcross = \_subsample\_proportional(x, mmsi[n1:n2], size//4)

    xtest = \_subsample\_proportional(x, mmsi[n2:], size//4)

    # except Exception, e:

    #     print "==== Broken data in the DataFrame ===="

    #     import pdb, sys

    #     sys.last\_traceback = sys.exc\_info()[2]

    #     pdb.set\_trace()

    return xtrain, xcross, xtest