Random Forest Classifier

The data for this analysis is of the Pima Indians and their liklihood of diabetetes. The data can be found here: https://data.world/uci/pima-indians-diabetes

This script implements a random forest classifier where the classifier will be evaluated via the out-of-bag (OOB) error estimate, using the above dataset.

Each tree in the forest is constructed using a different bootstrap sample from the original data.

ENJOY

In [9]:

```
import csv
import numpy as np
import ast
from datetime import datetime
from math import log, floor, ceil
```

In [10]:

```
class Utility(object):
    """
    Class with the necessary utility functions
    """
    def entropy(self, class_y):

        """
        This method compute the entropy for a list of classes

        Input:
           class_y         : list of class labels (0's and 1's)

        Output:
            entropy
        """

        entropy = 0
        val = np.mean(class_y)

        if val == 0:
            entropy =  entropy
        elif val == 1:
            entropy = entropy
        else:
            entropy = (-val)*np.log2(val) - (1-val) * np.log2(1-val)

        return entropy


    def partition_classes(self, X, y, split_attribute, split_val):

        """
        This method partitions the classes.
        First check if the split attribute is numerical or categorical
        If the split attribute is numeric, split_val should be a numerical value
        For example, your split_val could be the mean of the values of split_attribute
        If the split attribute is categorical, split_val should be one of the categori

        Inputs:
           X                : data containing all attributes
           y                : labels
           split_attribute : column index of the attribute to split on
           split_val       : either a numerical or categorical value to divide the spl

        """


        X_left = []
        X_right = []

        y_left = []
        y_right = []


        l = X[:,split_attribute] <= split_val
        r = X[:,split_attribute] > split_val

        X_left = X[l,:]
        X_right = X[r,:]

        y_left = y[l]
        y_right = y[r]

        return (X_left, X_right, y_left, y_right)



    def information_gain(self, previous_y, current_y):

        """
        This metho computes and returns the information gain from partitioning the pre
        into the current_y labels.

        Inputs:
           previous_y: the distribution of original labels (0's and 1's)
           current_y:  the distribution of labels after splitting based on a particula
                       split attribute and split value

        """

        info_gain = 0

        val = len(current_y[0])/len(previous_y)

        if val == 0:
            info_gain =  info_gain
        elif val == 1:
            info_gain = info_gain
        else:
            info_gain = Utility.entropy(self,previous_y) - Utility.entropy(self,curren

        return info_gain


    def best_split(self, X, y):

        """
        For each node find the best split criteria and return the
        split attribute, spliting value along with
        X_left, X_right, y_left, y_right (using partition_classes)

         Inputs:
           X       : Data containing all attributes
           y       : labels

        """

        split_attribute = 0
        split_value = 0
        X_left, X_right, y_left, y_right = [], [], [], []

        baseline = -1
        for b in range(len(X)):
            split_value = X[b][split_attribute]
            X_left, X_right, y_left, y_right = Utility.partition_classes(self,X,y,spli
            info_gain = Utility.information_gain(self,y,[y_left,y_right])

            if not np.isnan(info_gain) and info_gain > baseline:
                baseline = info_gain
                best_split_value = split_value


        return best_split_value, baseline




    def best_feature(self,X,y):
        """
        Gets best feature for the model.

        Inputs:
           X       : Data containing all attributes
           y       : labels

        """

        only_now = -1
        for b in range(len(X[0])):
            only_now_val, info_gain = Utility.best_split(self,X,y)
            if not np.isnan(info_gain) and info_gain >only_now:
                only_now = info_gain
                best_s_feature = b
                best_s_val = only_now_val
        return best_s_feature,best_s_val
```

In [11]:

```
class DecisionTree(object):
    def __init__(self, max_depth):
        """
        Initializing the tree as an empty list

        """
        self.tree = np.empty((0,4), int) #need list of list ie array to match input o
        self.max_depth = max_depth
        self.leaf_size = 10


    def learn(self, X, y, par_node = {}, depth=0):

        """
        This method trains the decision tree (self.tree) using the the sample X and la
        and the methods from the Utility class
        """


        if len(y) <= self.leaf_size:
            self.tree = np.append(self.tree, [["leaf", np.mean(y), np.nan, np.nan]], a
            self.addNext()
        elif (y == y[0]).all():
            self.tree = np.append(self.tree, [["leaf", y[0], np.nan, np.nan]], axis =
            self.addNext()
        else:
            temp = -1
            for i in range(X.shape[1]):
                SplitVal = np.median(X[:, i])
                X_left, X_right, y_left, y_right = Utility.partition_classes(self,X, y
                IG = Utility.information_gain(self,y, [y_left, y_right])
                if not np.isnan(IG) and IG > temp:
                    temp = IG
                    index = i
            SplitVal = np.median(X[:, index])
            X_left, X_right, y_left, y_right = Utility.partition_classes(self,X, y, in

            if len(X_left) == 0 or len(X_right) == 0:
                self.tree = np.append(self.tree, [["leaf", np.mean(y), np.nan, np.nan]
                self.addNext()
            else:
                self.tree = np.append(self.tree, [[index, SplitVal, 1, None]], axis =
                self.learn(X_left, y_left)
                self.learn(X_right, y_right)

    def addNext(self):

        """
        Method used to add the next value for the tree conditionally
        """
        n = self.tree.shape[0]
        for i in range(n):
            if self.tree[n-1-i, 3] == None:
                self.tree[n-1-i, 3] = i + 1
                break

    def classify(self, record):

        """
        Method to classify the record using self.tree and return the predicted label

        """

        curr = 0
        while True:
            if self.tree[curr, 0] == "leaf": #want to keep adding if the leaf is there
                return int(float(self.tree[curr, 1]) + 0.5)
            else:
                curr += self.tree[curr, 2] if record[self.tree[curr, 0]] <= self.tree
```

In [12]:

```
class RandomForest(object):

    """
    Here,
    1. X is assumed to be a matrix with n rows and d columns where n is the
    number of total records and d is the number of features of each record.
    2. y is assumed to be a vector of labels of length n.
    3. XX is similar to X, except that XX also contains the data label for each
    record.
    """

    num_trees = 0
    decision_trees = []

    # the bootstrapping datasets for trees
    # bootstraps_datasets is a list of lists, where each list in bootstraps_datasets
    bootstraps_datasets = []

    # the true class labels, corresponding to records in the bootstrapping datasets
    # bootstraps_labels is a list of lists, where the 'i'th list contains the labels
    # the 'i'th bootstrapped dataset.
    bootstraps_labels = []

    def __init__(self, num_trees):
        """
        Initialization done here
        """
        self.num_trees = num_trees
        self.decision_trees = [DecisionTree(max_depth=10) for i in range(num_trees)]
        self.bootstraps_datasets = []
        self.bootstraps_labels = []

    def _bootstrapping(self, XX, n):

        """
        This method creates a sample dataset of size n by sampling with replacement
        from the original dataset XX.

        """
        # Reference: https://en.wikipedia.org/wiki/Bootstrapping_(statistics)

        samples = [] # sampled dataset
        labels = []  # class labels for the sampled records

        #get XX to array
        XX = np.array(XX)
        decision = np.random.randint(0,n,size = n)

        samples = XX[:,:-1][decision]
        labels = XX[:,-1][decision]

        return (samples, labels)

    def bootstrapping(self, XX):
        """
        Initializing the bootstap datasets for each tree
        """

        for i in range(self.num_trees):
            data_sample, data_label = self._bootstrapping(XX, len(XX))
            self.bootstraps_datasets.append(data_sample)
            self.bootstraps_labels.append(data_label)

    def fitting(self):

        """
        This method trains `num_trees` decision trees using the bootstraps datasets
        and labels by calling the learn function from your DecisionTree class.
        """


        for b in range(self.num_trees):
            self.decision_trees[b].learn(self.bootstraps_datasets[b],self.bootstraps_l


    def voting(self, X):

        """
        This method votes for the best results via the following logic

               1. Find the set of trees that consider the record as an
                  out-of-bag sample.
               2. Predict the label using each of the above found trees.
               3. Use majority vote to find the final label for this recod.

        """
        y = []

        for record in X:
            votes = []

            for i in range(len(self.bootstraps_datasets)):
                dataset = self.bootstraps_datasets[i]

                if record not in dataset:
                    OOB_tree = self.decision_trees[i]
                    effective_vote = OOB_tree.classify(record)
                    votes.append(effective_vote)

            counts = np.bincount(votes)

            if len(counts) == 0:

                # Handle the case where the record is not an out-of-bag sample
                # for any of the trees.
                y = np.append(y, self.decision_trees[0].classify(record))
            else:
                y = np.append(y, np.argmax(counts))

        return y
```

In [13]:

```
# Initialize according to my implementation
# Ensure Minimum forest_size should be 10
forest_size = 10
```

Initialize the training: get timing and results to run

In [15]:

```
# start time
start = datetime.now()
X = list()
y = list()
XX = list()  # Contains data features and data labels
numerical_cols = set([i for i in range(0, 9)])  # indices of numeric attributes (colu

# Loading data set
print("Reading the data")
with open("../data/pima-indians-diabetes.csv") as f:
    next(f, None)
    for line in csv.reader(f, delimiter=","):
        xline = []
        for i in range(len(line)):
            if i in numerical_cols:
                xline.append(ast.literal_eval(line[i]))
            else:
                xline.append(line[i])

        X.append(xline[:-1])
        y.append(xline[-1])
        XX.append(xline[:])

# Initializing a random forest.
randomForest = RandomForest(forest_size)

# Creating the bootstrapping datasets
print("Creating the bootstrap datasets")
randomForest.bootstrapping(XX)

# Building trees in the forest
print("Fitting the forest")
randomForest.fitting()

# Calculating an unbiased error estimation of the random forest
# based on out-of-bag (OOB) error estimate.
y_predicted = randomForest.voting(X)

# Comparing predicted and true labels
results = [prediction == truth for prediction, truth in zip(y_predicted, y)]

# Accuracy
accuracy = float(results.count(True)) / float(len(results))

print("Accuracy: %.4f" % accuracy)
print("OOB estimate: %.4f" % (1 - accuracy))

# end time
print("Execution time: " + str(datetime.now() - start))
```

```
Reading the data
Creating the bootstrap datasets
Fitting the forest
Accuracy: 0.7604
OOB estimate: 0.2396
Execution time: 0:00:00.627372
```

In [ ]: