**Random forest**

**Code**

from csv import reader

from math import sqrt

from random import randrange, seed

def load\_csv(iris):

    """This method loads a csv file"""

    dataset = list()

    with open('iris.csv', 'r') as file:

        csv\_reader = reader(file)

        for row in csv\_reader:

            if not row:

                continue

            dataset.append(row)

    return dataset

def str\_column\_to\_float(dataset, column):

    """This method converts a string column to float"""

    for row in dataset:

        row[column] = float(row[column].strip())

def str\_columm\_to\_int(dataset, column):

    """This method converts a string column to int"""

    class\_values = [row[column] for row in dataset]

    unique = set(class\_values)

    lookup = dict()

    for i, value in enumerate(unique):

        lookup[value] = i

    for row in dataset:

        row[column] = lookup[row[column]]

    return lookup

def cross\_validation\_split(dataset, k\_folds):

    """This method splits a dataset into k folds"""

    dataset\_split = list()

    dataset\_copy = list(dataset)

    fold\_size = int(len(dataset) / k\_folds)

    for i in range(k\_folds):

        fold = list()

        while(len(fold) < fold\_size):

            index = randrange(len(dataset\_copy))

            fold.append(dataset\_copy.pop(index))

        dataset\_split.append(fold)

    return dataset\_split

def accuracy\_score(actual, predicted):

    """This method predicts the accuracy percentage"""

    correct = 0

    for i in range(len(actual)):

        if actual[i] == predicted[i]:

            correct += 1

    return correct / float(len(actual)) \* 100.0

def evaluate\_algorithm(dataset, algorithm, k\_folds, \*args):

    """This method evaluates the algorithm using a cross validation split"""

    folds = cross\_validation\_split(dataset, k\_folds)

    scores = list()

    for fold in folds:

        train\_set = list(folds)

        train\_set.remove(fold)

        train\_set = sum(train\_set, [])

        test\_set = list()

        for row in fold:

            row\_copy = list(row)

            test\_set.append(row\_copy)

            row\_copy[-1] = None

        predicted = algorithm(train\_set, test\_set, \*args)

        actual = [row[-1] for row in fold]

        accuracy = accuracy\_score(actual, predicted)

        scores.append(accuracy)

        return scores

def test\_split(index, value, dataset):

    """This method split a dataset based on an attribute and an attribute value"""

    left, right = list(), list()

    for row in dataset:

        if row[index] < value:

            left.append(row)

        else:

            right.append(row)

    return left, right

def gini\_index(groups, classes):

    """This method calculates the gini index for a split dataset"""

    # count all samples at split point

    n\_instances = float(sum([len(group) for group in groups]))

    # sum weighted gini index for each group

    gini = 0.0

    for group in groups:

        size = float(len(group))

        # avoid divide ny zero

        if size == 0:

            continue

        score = 0.0

        # score tje group based on the score for each class

        for class\_val in classes:

            p = [row[-1] for row in group].count(class\_val) / size

            score += p \* p

        # weight the group score by its relative size

        gini += (1.0 - score) \* (size / n\_instances)

    return gini

def get\_split(dataset, n\_features):

    """This method selects the best split for the dataset"""

    class\_values = list(set(row[-1] for row in dataset))

    b\_index, b\_value, b\_score, b\_groups = 999, 999, 999, None

    features = list()

    while len(features) < n\_features :

        index = randrange(len(dataset[0]) - 1)

        if index not in features:

            features.append(index)

    for index in features:

        for row in dataset:

            groups = test\_split(index, row[index], dataset)

            gini = gini\_index(groups, class\_values)

            if gini < b\_score:

                b\_index, b\_value, b\_score, b\_groups = index, row[index], gini, groups

    return {'index':b\_index, 'value':b\_value, 'groups':b\_groups}

def to\_terminal(group):

    """Create a terminal node value"""

    outcomes = [row[-1] for row in group]

    return max(set(outcomes), key=outcomes.count, default=None)

def split(node, max\_depth, min\_size, n\_features, depth):

    left, right = node['groups']

    del node['groups']

    # check for a no split

    if not left or not right:

        node['left'] = node['right'] = to\_terminal(left + right)

    # check for max\_depth

    if depth >= max\_depth:

        node['left'], node['right'] = to\_terminal(left), to\_terminal(right)

        return

    # process left child

    if len(left) <= min\_size:

        node['left'] = to\_terminal(left)

    else:

        node['left'] = get\_split(left, n\_features)

        split(node['left'], max\_depth, min\_size, n\_features, depth+1)

    # process right child

    if len(right) <= min\_size:

        node['right'] = to\_terminal(right)

    else:

        node['right'] = get\_split(right, n\_features)

        split(node['right'], max\_depth, min\_size, n\_features, depth+1)

def build\_tree(train, max\_depth, min\_size, n\_features):

    """This method builds a decision tree"""

    root = get\_split(train, n\_features)

    split(root, max\_depth, min\_size, n\_features, 1)

    return root

def predict(node, row):

    """This method makes a prediction with a decision tree"""

    if row[node['index']] < node['value']:

        if isinstance(node['left'], dict):

            return predict(node['left'], row)

        else:

            return node['left']

    else:

        if isinstance(node['right'], dict):

            return predict(node['right'], row)

        else:

            return node['right']

def subsample(dataset, ratio):

    """This method creates a random subsample from the dataset with replacement"""

    sample = list()

    n\_sample = round(len(dataset) \* ratio)

    while len(sample) < n\_sample:

        index = randrange(len(dataset))

        sample.append(dataset[index])

    return sample

def bagging\_predict(trees, row):

    """This method makes a prediction a list of bagged trees"""

    predictions = [predict(tree, row) for tree in trees]

    return max(set(predictions), key=predictions.count)

def random\_forest(train, test, max\_depth, min\_size, sample\_size, n\_trees, n\_features):

    """Random Forest Algorithm"""

    trees = list()

    for i in range(n\_trees):

        sample = subsample(train, sample\_size)

        tree = build\_tree(sample, max\_depth, min\_size, n\_features)

        trees.append(tree)

    predictions = [bagging\_predict(trees, row) for row in test]

    return predictions

"""Test run the algorithm"""

seed(2)

# load and prepare the data

iris = ""

dataset = load\_csv('iris.csv')

# convert string attributes to integers

for i in range(0, len(dataset[0]) - 1):

    str\_column\_to\_float(dataset, i)

# convert class columns to integers

str\_columm\_to\_int(dataset, len(dataset[0]) - 1)

# evaluate algorithm

k\_folds = 10

max\_depth = 10

min\_size = 1

sample\_size = 1.0

n\_features = int(sqrt(len(dataset[0]) - 1))

n\_trees = 20

scores = evaluate\_algorithm(dataset, random\_forest, k\_folds, max\_depth, min\_size, sample\_size, n\_trees, n\_features)

print("Trees: %d" % n\_trees)

print("Scores: {}" .format(scores))

print("Mean Accuracy: %.3f%%" % (sum(scores) / float(len(scores))))

**OUTPUT**

