Random forest Algorithm for pima diabetes dataset

Step1. Read csv file.

```
filename = 'pima-indians-diabetes.data.csv'
dataset = load_csv(filename)
```

Step2. Initialize parameters :

n_folds value of 5 was used for cross-validation, Deep trees were constructed with a max depth of 10 and a minimum number of training rows at each node of 1. n_features The number of features considered at each split point

```
n_folds = 5
max_depth = 10 # max depth of the tree
min_size = 1   #min size of the branches
sample_size = 1.0   #min size to create sub samples of the dataset
n_features = int(sqrt(len(dataset[0])-1))
```

Step3. Evaluate algorithm. Split the data into test and train set.

```
 # Split a dataset into n_folds
def cross_validation_split(dataset, n_folds):
 dataset_split = list()
 dataset_copy = list(dataset)
 fold_size = int(len(dataset) / n_folds)
 for i in range(n_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
```

Step4. Create sub samples of the dataset at random.

```
def subsample(dataset, ratio):
 sample = list()
 n_sample = round(len(dataset) * ratio)
 while len(sample) < n_sample:
        index = randrange(len(dataset))
        sample.append(dataset[index])
 return sample
```

Step5. Create trees of the sub samples. In the creation of each of the tree, Select the best split point for a dataset. To do that ,calculate the *gini* index for a split dataset. Gini index is a cost function that calculates the purity of the groups of data created by the split point.

```
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)
        return
    # 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)
```

Step6. Make a prediction with a decision tree for each sub sampled tree using bagging predict.

```
def predict(node, row):
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']
```

Step7. Store the returned predictions in a *predicted* list and actual values in *actual* list.

```
predictions = [bagging_predict(trees, row) for row in test]
return(predictions)

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

Step8. Using the actual and the predicted values calculate the accuracy, precision, recall and f1.

```
accuracy = accuracy_metric(actual, predicted)
precision = precision_score(actual,predicted)
recalls   = recall_score(actual,predicted)
F1 = recall_score(actual,predicted)
```