Implent your own Decision Tree/Random Forest!

In this python notebook, you will create a basic decision tree on pandas data, and train a classifier on the Iris dataset. Then, you will implement a type of bagging and create a random forest classifier!

First, import the required modules:

In [ ]:

```
from sklearn.datasets import load_iris
import pandas as pd
import numpy as np
```

Then import and preview the data:

In [ ]:

```
iris = load_iris()
df = pd.DataFrame(iris.data)
df['species'] = iris.target
df.head()
```

We have four features labeled 0, 1, 2, and 3. These stand for the length and the width of the sepals and petals, in centimeters. We want to use these four features to predict whether the species is one of three types of Iris plant, labeled 0, 1, or 2.

Now, we split the dataset into training and test samples.

In [ ]:

```
df['is_train'] = np.random.uniform(0, 1, len(df)) <= .75
train, test = df[df['is_train']==True], df[df['is_train']==False]
train = train.drop(['is_train'], axis = 1)
test = test.drop(['is_train'], axis = 1)
```

Disorder (Splitting Metric)

First, we want to implement some measure of disorder in a set of data.

Implement either information gain or GINI impurity discussed in class. (for reference the equations are in 189 notes here https://www.eecs189.org/static/notes/n25.pdf)

The argument `data` is a pandas dataframe containing the features and labels of several data points. We calculate disorder based on the labels, or the last column of the data. Note: make sure that you make this function work for different data (i.e. your function should work for data of different dimensions).

In [ ]:

```
def disorder(data):
    labels = data[data.columns[-1]]
    gini = 1
    if len(data) > 0:
        for _class in [0, 1, 2]:
            num = len(data[data['species'] == _class].index)
            p_i = num / len(data)
            gini -= p_i**2
    return gini
```

We now create a split function. This function takes in a dataset, and indices for a row and column. We then return two dataframes split on the `column` th feature. The left dataset should contain all of the data where the `column` th feature is greater or equal to the `column` th feature of the `row` th datapoint, and the right should contain the rest. Use the disorder metric you implemented in the function above.

In [ ]:

```
def split_on_row_column(data, row, column):
    threshold = data.iloc[row][column]
    left = data[data[column] >= threshold].drop(column, axis=1)
    right =  data[data[column] < threshold].drop(column, axis=1)
    return threshold, left, right
```

We now want to define our recursive tree class. During training, there are two cases for a node. If the data is all one label, the node is a leaf node, and we return this value during inference. If the data is not all the same label, we find the best split of the data by iterating through all of features and rows in the data. Use the split function defined above to find the best split.

Inference takes in a row of a pandas dataframes and returns the predicted class.

In [ ]:

```
class Node:
    def __init__(self, data, max_depth = 10):
        self.data = data
        self.total_count = len(data)
        self.predicted_class = -1
        self.max_depth = max_depth
        self.left = None
        self.right = None
        self.is_pure_leaf = False

    def train(self):
        self.predicted_class = self.data['species'].mode().iloc[0]
        # if the data for this node is completely homogenous, then it is a leaf node.
        if (disorder(self.data) == 0) or (self.max_depth == 0):

            # toggling pure leaf flag
            self.is_pure_leaf = True

        # if the data for this node is NOT completely homogenous, then find the best spli
        else:
            self.best_gini = 1
            self.best_feature, best_thresh = None, None

            features = self.data.copy().iloc[:, :-1].columns

            # iterating through possible features as thresholds
            for i, feature in enumerate(features):

                # iterating through actual thresholds
                for row_i in range(len(self.data[feature])):

                    # SPLIT BASED ON THRESHOLD
                    threshold, left, right = split_on_row_column(self.data, row_i, feature

                    # COMPUTE GINI OF SPLIT (WEIGHTED AVERAGE OF GINI)
                    left_gini = disorder(left) * len(left) / self.total_count
                    right_gini = disorder(right) * len(right) / self.total_count

                    split_gini = left_gini + right_gini

                    # SAVE BEST SPLIT
                    if split_gini < self.best_gini:
                        self.best_gini = split_gini
                        self.best_feature = feature
                        self.best_thresh = threshold

                        self.left = Node(left, self.max_depth - 1)
                        self.right = Node(right, self.max_depth - 1)


                self.left.train()
                self.right.train()

    def inference(self, x):
        if self.is_pure_leaf:
            return self.predicted_class
        else:
            if (x[self.best_feature] >= self.best_thresh):
                return self.left.inference(x)
            else:
                return self.right.inference(x)
```

Now initialize and train a decision tree:

In [ ]:

```
tree = Node(train, 10)
tree.train()
```

Note that we don't check the training accuracy here (why?). We now want to validate our tree on the test dataset:

In [ ]:

```
def validate(model, data):
    ct = 0
    corr = 0
    for i in range(test.shape[0]):
        data = test.iloc[i]
        ct += 1
        if model.inference(data) == data['species']:
            corr += 1
    return corr/ct
validate(tree, test)
```

Random Forest!

Now we will implement data bagging with a random forest! The set up is similar to a single tree. We pass in the data to the forest, along with hyperparameters `n`, `frac`, anbd `m`, which correspond to the number of trees, the fraction of the dataset to use in each bag, the number or percentage of random features (depending on your own implementation) selected at each possible split. Note that the difference between random forests and just bagging is that random forests select a random subset of features per bag while bagging assumes all features are present in each sample. A good estimate for m in a dataset with `num_features` is m = sqrt(`num_features`). In the inference step we tally the number of votes from each decision tree and return the label with the most amount of votes.

In [ ]:

```
class Forest:
    def __init__(self, data, n, frac, m):
        self.data = data
        self.n = n
        self.frac = frac

    def train(self):
        self.trees = []
        for i in range(self.n):
            #YOUR CODE HERE

    def inference(self, x):
        #YOUR CODE HERE
        return 0
```

Train and validate your forest!

In [ ]:

```
forest = Forest(train, 30, .5)
forest.train()
```

In [ ]:

```
validate(forest, test)
```