Machine Learning

Lecture 4

The BaggingClassifier basically works by randomizing the data set.

The next classifier we will look at, takes the randomization one step further and also randomizes the set of features used in splitting each node.

Specifically to build a RandomForestClassifier with B trees we select random bootstrap samples d_1, d_2, …, d_B from the data set X. Thus each d_i is a set of samples, with repetitions, from X, containing the same number of points as X.

For each of the bootstrap samples we build a tree but at each node we only allow a subset of the features to split on. The set of features allowed at each node is a random sample of some specified size m of all the features.

Usually instead of specifying the actual size, m, of the random subset, we specify a fraction of the number of features, to be used in the random sample to be used to split a node.

Specifically, suppose we are at a node in a tree (built on some bootstrap sample d_i). We select a random subset of all the features, of the appropriate size, and then from this subset we select the feature and the value that maximizes the Gini index of the split.

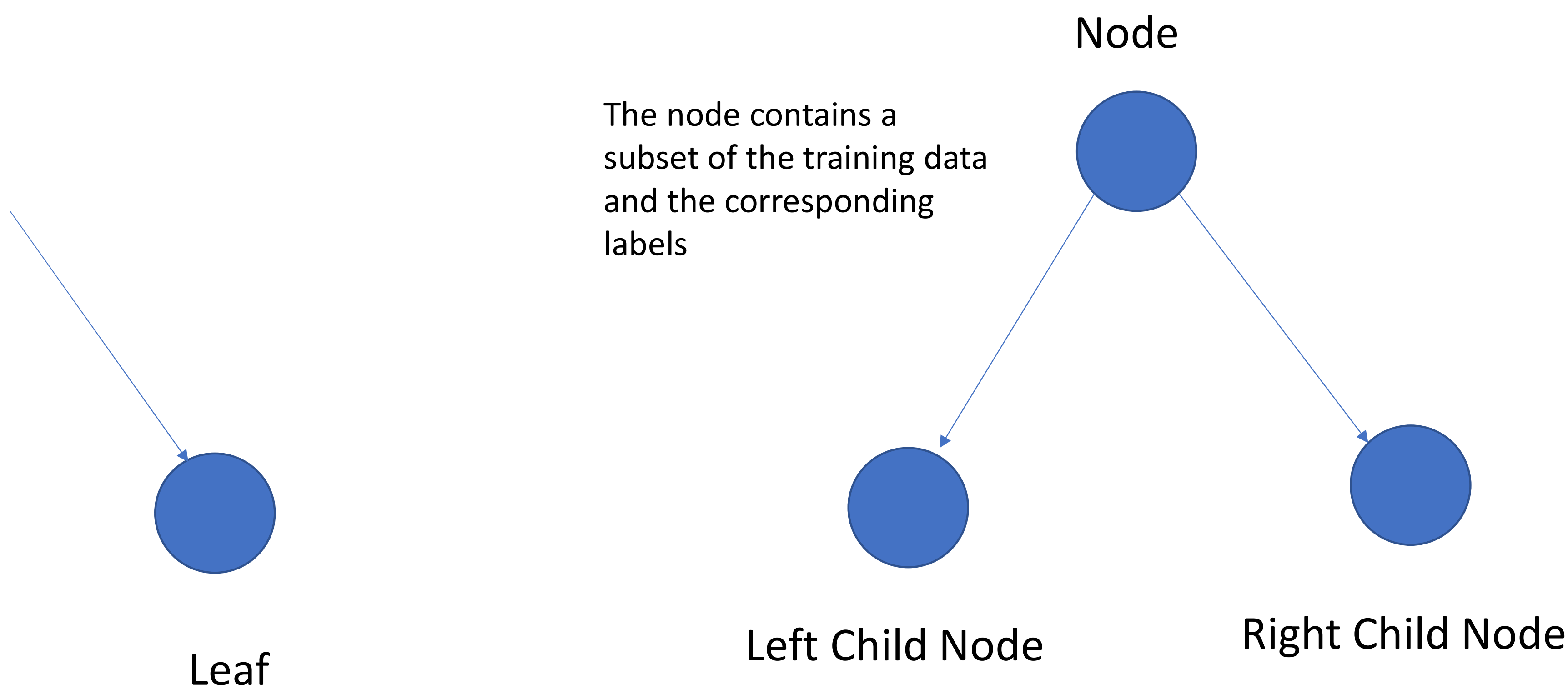
The prediction of a random forest is obtained by taking the prediction of each tree in the forest and then take the label with the most occurrences, i.e. the voting selection

We are going to program our own version of a random forest classifier, to get a thorough understanding of how the classifier works.

This is also an exercise in Object Oriented Programming

- A forest consists of trees.
- A tree consists of branches
- A branch consists of a node and two child nodes if it is not a leaf.
  If it is a leaf it has no child nodes

To build a forest class we start by building a node class.



The split divides the data into two subsets with their labels. The split is defined by a feature, f, and a threshold value, x, of that feature. The data in the left node is the set of data points d with d[f] < x

We define a Node class

We give the attributes of the class default values in the constructor.
We have to pay special attention to the min_samples_leaf feature.
A node with less than 2*min_samples_leaf cannot be split as that would result in at least one node containing less than min_samples_leaf data points. Thus if there are fewer than 2*min_leaf_samples we label it a Leaf

```python
class Node():
        def __init__(self,data:pd.DataFrame = None,labels=None,min_samples_leaf = 1,max_features=0.5):
            self.data = data
            self.labels = labels
            self.left = None
            self.right = None
            self.split = None
            self.max_features = max_features
            self.min_samples_leaf = min_samples_leaf
            if len(self.data) < 2 * self.min_samples_leaf:
                self.isLeaf = True
            else: self.isLeaf = False

        def gini_index(self,labels):

            C = Counter(labels)
            total = len(labels)
            frequencies = { k:v/total for (k,v) in C.items()}
            gini_index = np.sum([p*(1-p) for p in frequencies.values()])
            return gini_index
```

We split at a feature and a value that minimizes the Gini index of the split.

First we write a method to compute the Gini index of the node

```
def gini_index(self,labels):

    C = Counter(labels)
    total = len(labels)
    frequencies = { k:v/total for (k,v) in C.items()}
    gini_index = np.sum([p*(1-p) for p in frequencies.values()])
    return gini_index
```

Counter is part of the collections library. It takes a list and returns a dict of the number of times each item occurs in the list

```
from collections import Counter
```

```
root.labels
```

```
array([1, 2, 2, 2, 3, 1, 0, 2, 3, 1, 2, 2, 2, 0, 0, 0, 0, 3, 0, 1, 2, 0,
       2, 3, 2, 2, 2, 2, 1, 3, 1, 1, 1, 1, 1, 1, 2, 3, 0, 2, 1, 1, 0, 3,
       2, 2, 3, 1, 3, 2, 0, 0, 1, 3, 0, 1, 0, 2, 0, 2, 3, 3, 1, 3, 3, 0,
       3, 3, 3, 0, 1, 0, 2, 0, 1, 2, 3, 3, 2, 2, 1, 2, 0, 0, 1, 3, 1, 0,
       3, 2, 1, 3, 3, 2, 2, 3, 3, 3, 0, 1, 3, 1, 3, 1, 3, 1, 0, 2, 0, 1,
       3, 2, 3, 3, 1, 3, 1, 3, 0, 3, 2, 3, 2, 3, 3, 2, 0, 0, 1, 2, 2, 3,
       0, 0, 2, 0, 0, 3, 1, 2, 2, 1, 0, 0, 2, 1, 3, 1, 3, 2, 3, 0, 2, 3,
       3, 3, 3, 0, 3, 2, 3, 2, 0, 0, 2, 1, 1, 3, 0, 3, 3, 2, 2, 3, 1, 1,
       1, 2, 3, 0, 1, 0, 3, 3, 2, 2, 0, 3, 1, 2, 1, 3, 0, 3, 1, 2, 1, 2,
       1, 2, 2, 0, 2, 1, 0, 1, 3, 1, 1, 0, 3, 2, 0, 2, 3, 3, 0, 1, 1, 0,
       2, 2, 0, 1, 0, 2, 0, 2, 0, 3, 2, 3, 1, 0, 0, 1, 3, 1, 2, 0, 2, 2,
       0, 1, 2, 0, 3, 2, 2, 1, 3, 1, 2, 3, 0, 1, 3, 3, 0, 1, 1, 3, 0, 0,
       1, 1, 0, 0, 2, 0, 0, 0, 1, 0, 2, 2, 3, 3, 2, 2, 1, 2, 1, 0, 0, 0,
       3, 1, 0, 2, 1, 2, 0, 0, 3, 3, 3, 0, 0, 1, 2, 3, 0, 2, 0, 0, 2, 1,
       1, 1, 2, 2, 2, 3, 0, 2, 0, 3, 2, 1, 1, 1, 3, 0, 2, 2, 1, 2, 2, 0,
       1, 3, 2, 3, 0, 3, 0, 0, 2, 0, 2, 1, 2, 0, 3, 0, 1, 3, 3, 0, 3, 0,
       3, 0, 0, 2, 1, 1, 3, 3, 3, 3, 3, 0, 0, 2, 3, 2, 1, 3, 0, 0, 0, 0,
       0, 0, 0, 3, 3, 0, 3, 2, 1, 2, 0, 0, 2, 1, 3, 1, 3, 1, 1, 1, 0, 1,
       1, 1, 2, 3, 0, 2, 3, 0, 3, 2, 2, 2, 3, 2, 0, 0, 1, 1, 2, 2, 3, 0,
       0, 1, 2, 2, 1, 2, 1, 1, 2, 1, 2, 3, 1, 3, 2, 0, 3, 0, 3, 2, 3, 1,
       1, 1, 0, 1, 3, 3, 3, 3, 2, 1, 2, 0, 2, 1, 3, 1, 1, 2, 0, 2, 1, 1,
       2, 0, 3, 1, 2, 3, 0, 1, 3, 1, 1, 0, 1, 0, 0, 1, 1, 2, 3, 0, 3, 1,
       3, 2, 3, 0, 2, 1, 0, 0, 1, 0, 1, 3, 0, 1, 1, 3])
```

```
Counter(root.labels)
```

```
Counter({1: 123, 2: 124, 3: 126, 0: 127})
```

Counter can also get the most common item

```
1  Counter(root.labels).most_common()
```

```
[(0, 127), (3, 126), (2, 124), (1, 123)]
```

```
1  Counter(root.labels).most_common()[0][0]
```

```
0
```

If we want the actual item we can take the 0’th item in this list which is the tuple (0,127) and so we take the 0’th item of this tuple

We then compute the frequencies of each item by dividing the number of occurrences with the length of the list

```
frequencies = { k:v/total for (k,v) in C.items()}
frequencies
```

```
{1: 0.246, 2: 0.248, 3: 0.252, 0: 0.254}
```

```
frequencies.values()
```

```
dict_values([0.246, 0.248, 0.252, 0.254])
```

```
gini_index = np.sum([ p * (1 - p) for p in frequencies.values()])
gini_index
```

```
0.74996
```

Next we write the method to find the best split. Here we have to take into account the randomization of the features can select from.

```
def find_best_split(self):

    data_set = self.data
    labels = self.labels
    features = np.random.choice(data_set.columns,int(self.max_features * len(data_set.columns)))
```

To see how this works we use our shap_cols from last time

```
shap_cols
```

```
array(['fcf_yield', 'oiadpq', 'rd_saleq', 'market_cap', 'oancfy_q',
       'oeps12', 'ibadj12', 'book_value_yield', 'oepf12', 'quick_ratioq',
       'short_debtq', 'cf_yield', 'sic_6798', 'ibcy', 'xidoy', 'chechy',
       'inv_turnq', 'ocf_lctq', 'dpcq', 'cfmq', 'oancfy', 'roeq',
       'cfo-per-share', 'txpq', 'opmbdq', 'psq', 'dltisy', 'dltry',
       'yearly_sales', 'xsgay', 'lagppent4', 'fcf_ocfq', 'cash_ratioq',
       'de_ratioq', 'sale_equityq', 'evmq', 'opepsq', 'dvy', 'actq',
       'capxq', 'ceq4', 'fcf_csfhdq', 'sale_invcapq', 'apq', 'dlttq',
       'rect_actq', 'capeiq', 'nopiq', 'capxy', 'npmq', 'invt_actq',
       'oibdpy', 'accrualq', 'int_totdebtq', 'txtq', 'gpmq',
       'aftret_invcapxq', 'dpq'], dtype=object)
```

Optimal Features from Bagging notebook

```
optim_feats = np.array(['fcf_yield', 'oiadpq', 'rd_saleq', 'market_cap', 'oancfy_q',
       'oeps12', 'ibadj12', 'book_value_yield', 'oepf12', 'quick_ratioq',
       'short_debtq', 'cf_yield', 'sic_6798', 'ibcy', 'xidoy', 'chechy',
       'inv_turnq', 'ocf_lctq', 'dpcq', 'cfmq', 'oancfy', 'roeq',
       'cfo-per-share', 'txpq', 'opmbdq', 'psq', 'dltisy', 'dltry',
       'yearly_sales', 'xsgay', 'lagppent4', 'fcf_ocfq', 'cash_ratioq',
       'de_ratioq', 'sale_equityq', 'evmq', 'opepsq', 'dvy', 'actq',
       'capxq', 'ceq4', 'fcf_csfhdq', 'sale_invcapq', 'apq', 'dlttq',
       'rect_actq', 'capeiq', 'nopiq', 'capxy', 'npmq', 'invt_actq',
       'oibdpy', 'accrualq', 'int_totdebtq', 'txtq', 'gpmq',
       'aftret_invcapxq', 'dpq'], dtype=object)
```

```
len(optim_feats)
```

```
5]: 58
```

```
max_features = 0.5
features = np.random.choice(optim_feats,int(0.5 * len(optim_feats)),replace=False)
features
```

```
6]: array(['ocf_lctq', 'dlttq', 'inv_turnq', 'lagppent4', 'sale_equityq',
       'sale_invcapq', 'nopiq', 'yearly_sales', 'cfmq', 'gpmq', 'actq',
       'dpq', 'oeps12', 'sic_6798', 'evmq', 'roeq', 'quick_ratioq',
       'capxy', 'book_value_yield', 'capxq', 'ibadj12', 'opmbdq',
       'fcf_ocfq', 'xsgay', 'opepsq', 'npmq', 'txtq', 'int_totdebtq',
       'xidoy'], dtype=object)
```

```
len(features)
```

```
7]: 29
```

Our random subset of features contains about 0.5 times the total number of features, this parameter can be set to any number <= 1

Here we do not want to sample with replacement so we set replace=False

The subset of features will change whenever we run the method i.e. it will be different for different nodes

```
max_features = 0.5
features = np.random.choice(optim_feats,int(0.5 * len(optim_feats)),replace=False)
features
```

```
array(['ibadj12', 'book_value_yield', 'oancfy', 'sale_invcapq', 'txtq',
       'roeq', 'oiadpq', 'market_cap', 'npmq', 'rect_actq',
       'int_totdebtq', 'opepsq', 'oancfy_q', 'xsgay', 'lagppent4',
       'capeiq', 'dpcq', 'xidoy', 'quick_ratioq', 'gpmq', 'nopiq', 'dpq',
       'oeps12', 'oepf12', 'actq', 'dlttq', 'aftret_invcapxq', 'dltry',
       'ibcy'], dtype=object)
```

```
max_features = 0.5
features = np.random.choice(optim_feats,int(0.5 * len(optim_feats)),replace=False)
features
```

```
array(['sale_equityq', 'capeiq', 'ceq4', 'cf_yield', 'market_cap',
       'oibdpy', 'chechy', 'cfmq', 'capxq', 'oeps12', 'rd_saleq',
       'oancfy_q', 'sic_6798', 'txpq', 'gpmq', 'actq', 'evmq', 'npmq',
       'rect_actq', 'dpq', 'psq', 'quick_ratioq', 'de_ratioq', 'txtq',
       'dlttq', 'dltisy', 'xsgay', 'book_value_yield', 'lagppent4'],
      dtype=object)
```

We sample a random set of features to split on of size max_features * size of features. We are not sampling with replacement so we set replace=False

```
data_set = self.data
labels = self.labels
features = np.random.choice(data_set.columns,
                            int(self.max_features * len(data_set.columns)),
                            replace=False)
```

With the set of features selected we loop through all the selected features and for each feature through all the values of that feature

We initialize the values best_split , this will eventually become the Gini index of the best possible split.
We also initialize best_feature and best_value

```
best_split = np.inf

best_feature = None
best_value = None

for f in features:

    for x in data_set[f]:
```

Given the feature, f, and the value, x, we can now split the data_set. A data point z (a row in the data set) goes into the left node if it's value at feature f, z[f]< x and into the right node if z[f] >= x. We actually collect the indices of the data points in each split so we can also get the corresponding labels

```
L_idxs = data_set.index[data_set[f] < x].tolist()
R_idxs = data_set.index[data_set[f] >= x].tolist()


L_labels = labels[L_idxs]
R_labels = labels[R_idxs]
```

```
f = 2
x = data_set[2].iloc[234]
x
```

```
-0.6598871081669752
```

```
L_idxs = data_set.index[data_set[f] < x].tolist()
R_idxs = data_set.index[data_set[f] >= x].tolist()
```

```
L_idxs
```

```
302,
305,
306,
```

```
R_idxs
```

```
[0,
 1,
 3,
 5,
 6,
 8,
 9,
 10,
 13,
 14,
 18,
 19,
 20,
 21,
 22,
 23,
 24,
 26,
```

Both splits must have at least min_samples_leaf data points so if either of the two splits are too small we simply go to try the next value in the loop.

Otherwise we go on to compute the Gini index of the split

```
if (len(L_labels) < self.min_samples_leaf) or (len(R_labels) < self.min_samples_leaf):
    continue

else:

    gi_L = self.gini_index(L_labels)
    gi_R = self.gini_index(R_labels)

    gini_index_of_split = len(L_idxs) * gi_L + len(R_idxs) * gi_R
```

```python
 L_labels = labels[L_idxs]
R_labels = labels[R_idxs]
```

```python
def gini_index(labels):

            C = Counter(labels)
            total = len(labels)
            frequencies = { k:v/total for (k,v) in C.items()}
            gini_index = np.sum([p*(1-p) for p in frequencies.values()])
            return gini_index
```

```python
print(gini_index(L_labels))
print(gini_index(R_labels))

```

```
0.745986920332937
0.7493896484375
```

```python
gini_index_of_split = (len(L_labels)*gini_index(L_labels) + len(R_labels)*gini_index(R_labels))/len(data_set)
print(gini_index_of_split)
```

```
0.7486002155172414
```

If the new gini_index_of_split is better (i.e. <) than the previous best_split, we set the best_split to the new value and adjust all the values.

If it is not better we just fall through all the statements and continue the loop

```
if gini_index_of_split < best_split:
    best_split = gini_index_of_split
```

If we have a new best_split we set the left and right child nodes

```
self.left = Node(data=data_set.iloc[L_idxs].reset_index(drop=True),
                 labels = L_labels,min_samples_leaf = self.min_samples_leaf)

if (gi_L == 0) or (len(L_labels) < 2 * self.min_samples_leaf):
        self.left.isLeaf = True



self.right = Node(data = data_set.iloc[R_idxs].reset_index(drop=True),
                  labels = R_labels,min_samples_leaf = self.min_samples_leaf)

if (gi_R == 0) or (len(L_labels) < 2 * self.min_samples_leaf):
           self.right.isLeaf = True


best_feature = f
best_value = x

self.split = (best_feature,best_value)
```

If there are fewer than 2*min_samples_leaf in either of the child nodes that node cannot be split further so is a leaf (this is actually also done in the constructor so it is not really necessary)

```
data_set.iloc[L_idxs]
```

The index is messed up, we want the rows to be indexed by 0,1,2,3,4,.. so we fix the indices in the data sets for each of the child nodes

	fcf_yield	chechy	gpmq	cf_yield	nopiq	evmq	efftaxq	rect_turnq	pay_tu
1	0.071303	-481.393	0.277405	0.087732	9.514	-19.342331	-0.172719	3.122523	0.00
2	-0.001702	13.995	0.000000	0.001509	-7.474	0.000000	0.000000	0.000000	0.00
3	-0.408775	-40.011	0.000000	0.016696	-18.151	0.000000	0.000000	0.000000	0.00
5	-0.004927	-12.035	0.000000	-0.004927	-2.146	-74.002873	-0.007942	0.000000	0.00
7	0.052195	14.931	0.486650	0.116649	4.420	8.894997	0.383921	12.047315	11.58
10	0.023621	-124.031	0.000000	0.136287	-13.185	0.000000	0.000000	0.000000	0.00
13	0.058771	-1044.505	0.430208	0.576943	86.843	3.403872	0.248496	21.707243	1.52
14	-0.072626	1.022	0.163559	-0.038361	1.823	18.687658	0.377020	150.729448	29.87
15	0.008639	1.100	0.000000	0.044258	14.000	0.000000	0.000000	0.000000	0.00
17	0.000823	-17.532	0.170679	0.050788	1.171	15.041784	0.317015	217.683278	9.37

```
data = data_set.iloc[R_idxs].reset_index(drop=True)
```

We use the reset_index command on the DataFrame, remark that if we don’t use the drop=True, option we would get an extra column, namely the old index

```
1  data_set.iloc[L_idxs].reset_index()
```

Extra column consisting of the old index

	index	fcf_yield	chechy	gpmq	cf_yield	nop
0	1	0.071303	-481.393	0.277405	0.087732	9.5
1	2	-0.001702	13.995	0.000000	0.001509	-7.47
2	3	-0.408775	-40.011	0.000000	0.016696	-18.15
3	5	-0.004927	-12.035	0.000000	-0.004927	-2.14
4	7	0.052195	14.931	0.486650	0.116649	4.4

```
1  data_set.iloc[L_idxs].reset_index(drop=True)
```

Correct indexing

	fcf_yield	chechy	gpmq	cf_yield	nopiq	evr
0	0.071303	-481.393	0.277405	0.087732	9.514	-19.3423
1	-0.001702	13.995	0.000000	0.001509	-7.474	0.0000
2	-0.408775	-40.011	0.000000	0.016696	-18.151	0.0000
3	-0.004927	-12.035	0.000000	-0.004927	-2.146	-74.0028
4	0.052195	14.931	0.486650	0.116649	4.420	8.8949
5	0.023621	-124.031	0.000000	0.136287	-13.185	0.0000

```
self.left = Node(data=data_set.iloc[L_idxs].reset_index(drop=True),labels = L_labels)

if (gi_L == 0) or (len(L_labels) <= self.min_leaf_samples):
        self.left.isLeaf = True
```

The first line sets the left child node to the new node constructed from the split.
In the second line we check to see if the new node has Gini index = 0 which means it is pure, all the labels are the same. In this case the we do not want to further split the node i.e. it is a Leaf.
If the node has size < 2* min_leaf_samples the we don’t want to split it further so it is also a Leaf and we set the isLeaf attribute to True

Finally we store the feature and the threshold and store it in the Node's split attribute.

```
best_feature = f
best_value = x

self.split = (best_feature,best_value)
```

The `else: continue` is the else statement of the if statement

```
if gini_index_of_split < best_split:
    best_split = gini_index_of_split
```

so if the Gini index is not better than the previous best don’t make child nodes but skip to the next value in the feature column

This finishes the Node class

Next step in building a random forest is to write a Tree class

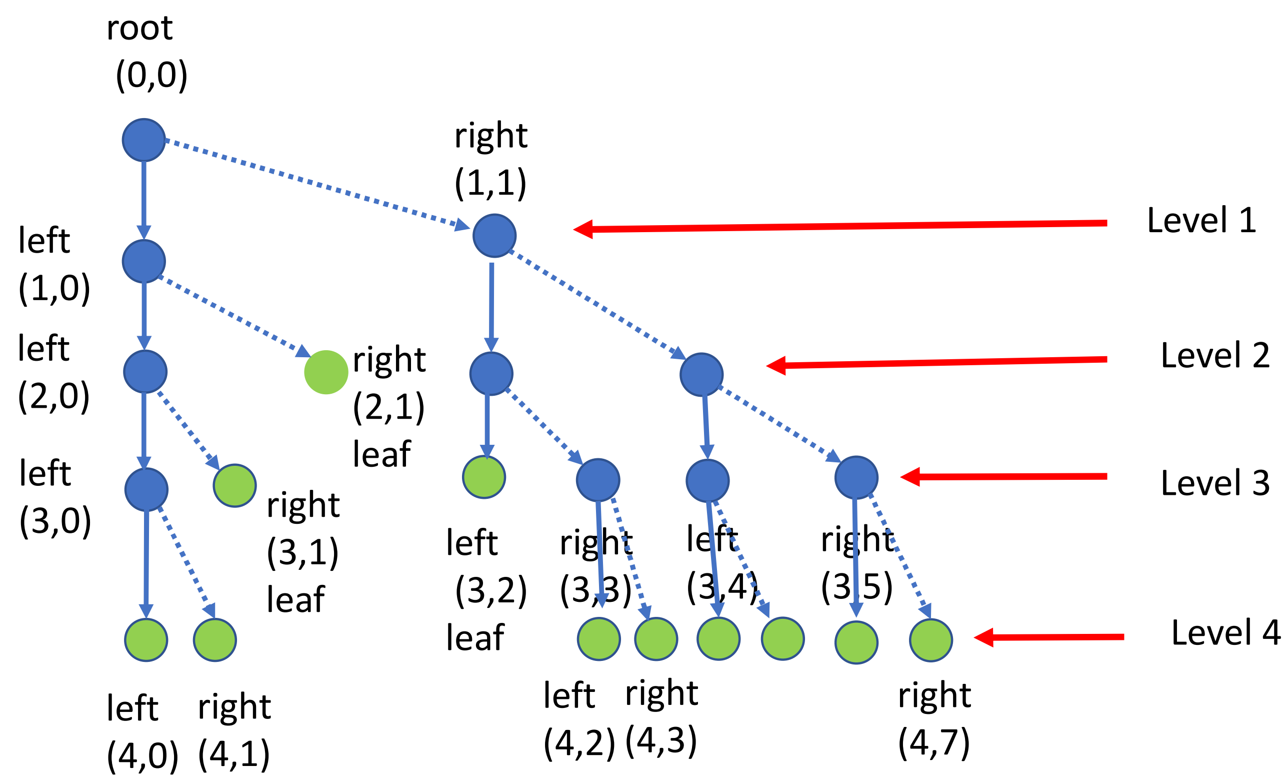
Remark that we pass the min_samples_leaf to the root node. The Node class method find_best_split will then pass it on to all the nodes in the tree

```
class Tree():

    def __init__(self,root,max_depth = np.inf,min_samples_leaf=1):
        self.root = root
        self.levels = [[self.root]]
        self.max_depth = max_depth
        self.splits = []
        self.min_samples_leaf = min_samples_leaf
        self.root.min_samples_leaf = self.min_samples_leaf

        self.build_tree()

```

So a tree object has a root Node, and a max_depth which we give the default value inf i.e. we do not initially put a limit of how deep the tree can get.

The tree also has a 2-dimensional array of nodes and a list of the splits, each split is a feature and a value

When we instantiate a tree object we run the method build_tree. To understand how this method works we can visualize an example of a nodes array



The way we are going to populate the nodes array is to fill in the levels sequentially. We start with the root node and sequentially fill in the levels with the child nodes constructed by splitting nodes

The list self.levels is a list of levels. self.levels[-1] is the last level we have constructed so at the first step it just contains the list consisting of the root node

```
def build_tree(self):

        while np.any([((n != None) and (not n.isLeaf)) for n in self.levels[-1]]) and (len(self.levels) < self.max_de

            level = []

            for node in [n for n in self.levels[-1] if ((n != None) and (not n.isLeaf))]:

                if (node != None) and (not node.isLeaf):

                    node.find_best_split()
                    level.append(node.left)
                    level.append(node.right)
                    self.splits.append(node.split)

                else: continue
```

We check that there are nodes that are not leaves

```
while np.any([((n != None) and (not n.isLeaf)) for n in self.levels[-1]]) and (len(self.levels) < self.max_depth):
```

and that the number of levels does not exceed the max_depth

```
and (len(self.levels) < self.max_depth):
```

Then we initialize an empty list and run through each of the nodes in the last level we have constructed

```
for node in [n for n in self.levels[-1] if ((n != None) and (not n.isLeaf))]:
```

This is the previous level that we have already populated

We check that the node is not empty and is not a leaf

```
if (node != None) and (not node.isLeaf):
```

If the node is not a leaf we can split it and append the child nodes to the level and the split, i.e. the feature we split at and the threshold, to the list self.splits

```
node.find_best_split()
level.append(node.left)
level.append(node.right)
self.splits.append(node.split)
```

If the node is either empty or is a leaf we continue to the next node

```
else: continue
```

Finally if the new level is empty we return and we are finished constructing the tree, otherwise we add the level to the self.nodes list and go back to the

```
while np.any([((n != None) and (not n.
```

statement.

```
if (len(level)==0):
        return
else:
        self.nodes.append(level)
```

Finally we append the populated level to the self.levels array

```
self.levels.append(level)
```

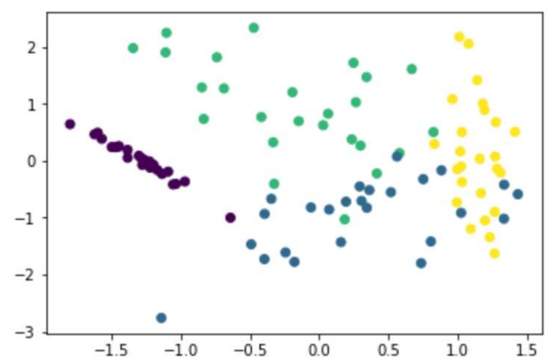
Let’s apply this to a data set

```
X,y = make_classification(n_classes=4,n_clusters_per_class=1,n_features=4)
```

```
import matplotlib.pyplot as plt
%matplotlib inline
```

```
plt.scatter(X[:,0],X[:,1],c=y)
```

```
]: <matplotlib.collections.PathCollection at 0x25ced6d2c88>
```




```
X = pd.DataFrame(data = X, columns = [0,1,2,3])
```

```
y
```

```
]: array([2, 1, 0, 0, 2, 1, 2, 3, 0, 3, 2, 1, 0, 1, 3, 3, 1, 0, 1, 3, 3, 1,
         1, 0, 2, 2, 3, 2, 1, 2, 3, 3, 0, 0, 3, 3, 3, 2, 1, 0, 0, 0, 0, 2,
```

```python
tree = Tree(root)
```

```python
len(tree.levels)
```

```
12
```

```python
len(tree.levels[8])
```

```
30
```

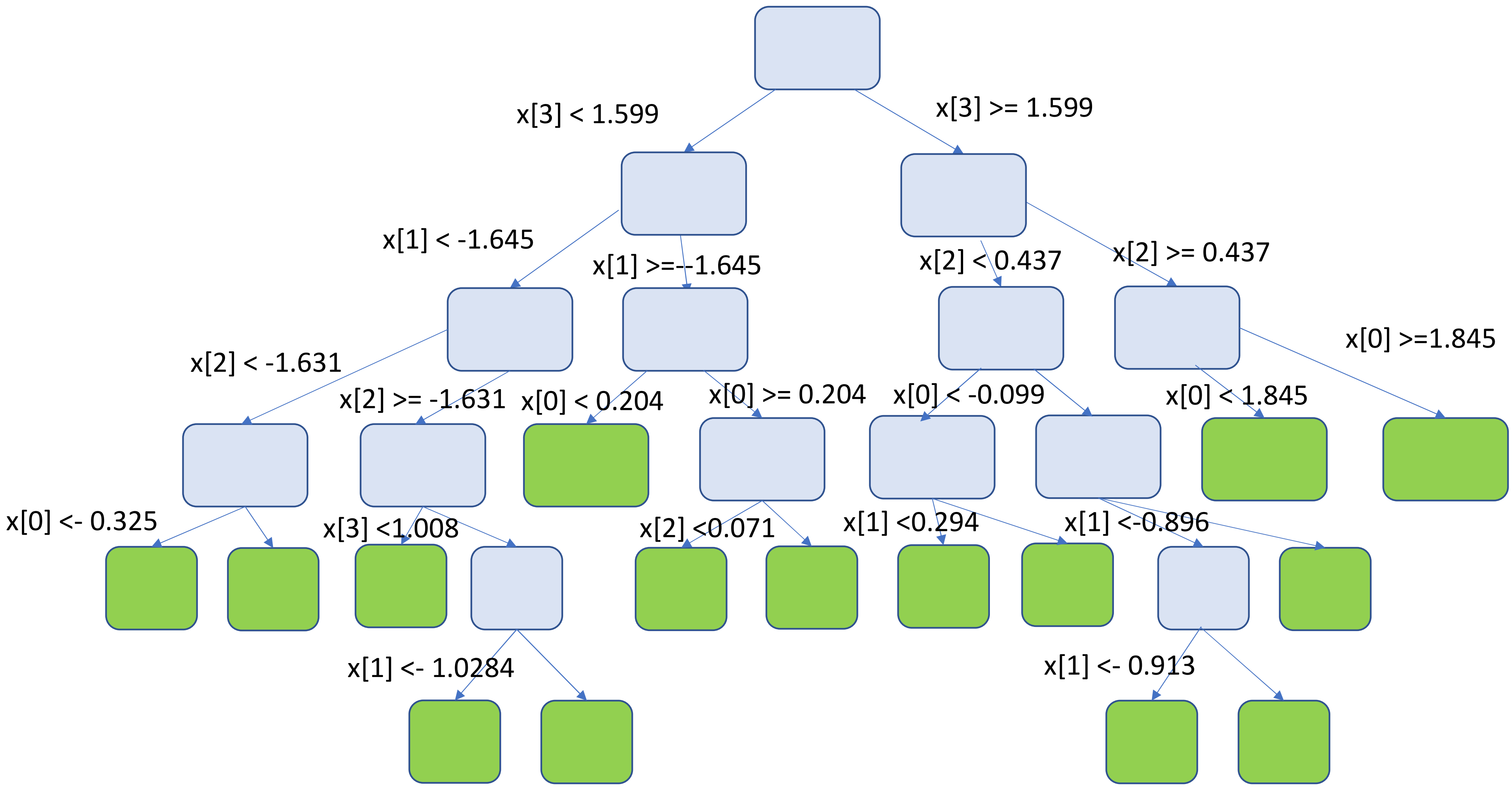
```python
tree.levels
```

```
[[<__main__.Node at 0x24afd8a8760>],
 [<__main__.Node at 0x24afd7f4b20>, <__main__.Node at 0x24afd6670a0>],
 [<__main__.Node at 0x24afd8b4d60>,
  <__main__.Node at 0x24afd8b4e20>,
  <__main__.Node at 0x24afd8b4c10>
```

The items in the array are pointers to the Node objects i.e. the addresses in memory of the objects

```
tree.splits
```

```
[(3, 1.5990850576679354),
 (1, -1.645307171542324 2),
 (2, 0.43735583490260144),
 (2, -1.631321781446071),
 (0, 0.20375409638001174),
 (0, -0.1479585857837894),
 (0, 1.8450854366158258),
 (2, -0.099664211843697),
 (0, 1.1112635957507992),
 (0, -0.34621361176733645),
 (2, 0.18648108613401282),
 (1, -1.0334712899157328),
 (1, 1.1442399765837272),
 (0, -1.9160791737603964),
 (1, -0.5757898586164623),
 (0, 0.781973250774354),
 (1, -0.09957294905049219),
 (1, 1.3143352275997173),
 (1, 0.49669756746190524),
 (2, -0.40910296761085885),
 (0, 0.8415315571130233),
 (2, 0.7773281318150193)
```



x[3] < 1.599
x[3] >= 1.599
x[1] < -1.645
x[1] >=--1.645
x[2] < 0.437
x[2] >= 0.437
x[2] < -1.631
x[2] >= -1.631
x[0] < 0.204
x[0] >= 0.204
x[0] < -0.099
x[0] < 1.845
x[0] >=1.845
x[0] <- 0.325
x[3] <1.008
x[2] <0.071
x[1] <0.294
x[1] <-0.896
x[1] <- 1.0284
x[1] <- 0.913

When the tree has been built we want to use it to make predictions. This means taking any data point, send it through the tree and decide which leaf it ends up in. The prediction is then the majority label in the leaf (remark that a leaf node may not be pure if we limit the max_depth or the min_leaf_samples > 1).

Here is the code for the predict method

```
def predict(self,x):

    idx = 0
    depth = len(self.levels)

    val = Counter(self.root.labels).most_common()[0][0]



    for i in range(depth):

            node = self.levels[i][idx]

            if (node.split != None):

                s = node.split

                if (x[s[0]] < s[1]):
                      if node.left in self.nodes[i+1]:
                        idx = self.levels[i+1].index(node.left)
                        val = Counter(node.left.labels).most_common()[0][0]

                else:
                      if node.right in self.nodes[i+1]:
                            idx = self.levels[i+1].index(node.right)
                            val = Counter(node.right.labels).most_common()[0][0]

            else: return val

    return val
```

We are going to find the level and the index in the level, of the node the data point ends up in.

We initialize the idx to 0 and find the depth of the tree i.e. how many levels it has. The value at the root node is the majority label

```
idx = 0
depth = len(self.levels)

val = Counter(self.root.labels).most_common()[0][0]
```

Then we begin to loop through the levels. For i=0 and idx =0 we are at the root

```
for i in range(depth):

        node = self.levels[i][idx]
```

If the node is not a leaf we find the split of the node s = (feature,value)

If x[feature] < value, x moves to the left child node and we find the index of the left child node in the level array = self.levels[i+1].

The val at this stage is then the majority label for this node.

If x[feature] >= value it moves to the right child node

```
if (node != None) and (not node.isLeaf):
    s = node.split
    if (s != None):

        if (x[s[0]] < s[1]):
            if node.left in self.nodes[i+1]:
                idx = self.nodes[i+1].index(node.left)
                val = Counter(node.left.labels).most_common()[0][0]

        elif (x[s[0]] >= s[1]):
            if node.right in self.nodes[i+1]:
                idx = self.nodes[i+1].index(node.right)
                val = Counter(node.right.labels).most_common()[0][0]
```

We then check to see if the node at level i and index idx has a non-empty split. This is the case precisely if we can split the node.

Node.split = s = (feat,thshld). If the value of the data point x at the feature feat = s[0] is < than the threshold thshld = s[1], the point goes to the left child node, node.left. This node is in the next level, i+1, and we find it's index in level i+1. This index now becomes the new value for idx

```
if (x[s[0]] < s[1]):
    if node.left in self.nodes[i+1]:
        idx = self.nodes[i+1].index(node.left)
```

Finally we compute the value at this node and set val to this value

```
val = Counter(node.left.labels).most_common()[0][0]
```

The other possibility, if x[s[0]] is not < s[1], (so x[s[0]] >= s[1]) sends x to the right child node and we do the same in this case.

After the first pass through the loop i = 1 and idx is either 0 or 1 depending on whether x went left or right.

At each step in the loop we find the level and the index of the node x went to until either we come to a leaf or we reach level = max_depth.

If we reach a leaf i.e. node.split = None before we have run all the way through the loop we terminate the method and return val

```
else: return val
```

If we run all the way through the loop the val is the terminal value of x and we return val

```
return val
```

We can try out the Tree class on the first of the three data sets

The root node is the node containing all the data and all the labels

```
root = Node(X,y)
```

This constructs the tree and we see it has depth 7

```
tree = Tree(root)
```

```
len(tree.nodes)
```

```
7
```

The nodes in the last level are all leaves

```
len(tree.nodes[6])
```

```
6
```

```
tree.nodes[6][0].isLeaf
```

```
True
```

The root node splits on feature 2 with threshold -0.547089…

```
tree.splits
```

```
[(2, -0.5470893221852209),
 (3, 0.7329516431003351),
 (3, 0.5714490832253741),
```

We check out the predict method on an arbitrary data point

```
x = X.iloc[30].values
```

```
x
```

```
array([ 0.65551171, -1.09283649, -0.98432888, -0.89738884])
```

```
tree.predict(x)
```

```
0
```

```
y[30]
```

```
0
```

The RandomForest class is now easy to code

```python
class RandomForestClassifier():

    def __init__(self,n_estimators=10,max_features=1.0,min_samples_leaf=1,bootstrap=False,max_depth = np.inf,
                 random_state=None):
        self.n_estimators = n_estimators
        self.max_features = max_features
        self.min_samples_leaf = min_samples_leaf
        self.max_depth = max_depth
        self.bootstrap = bootstrap
        if random_state is not None:
            np.random.seed(random_state)
        self.trees = []
```

We specify how many trees we want in the forest: n_estimators

The fraction of the features we will use in each split: max_features

The minimum number of data points in each node: min_samples_leaf

Whether we bootstrap from the data set: bootstrap

We can set a random seed, which means that the random sets will be the same every time we run the forest. Otherwise we will get different results at different runs

To make our RandomForestClassifier behave like the sklearn class, we define a fit method

We create the trees in the forest, if we bootstrap we select a bootstrap sample from the data set and make a tree with the bootstrapped data set in the node

If we don't bootstrap we make a tree with the full data set in the root node

We instantiate the tree and append it to the list of trees

```python
    def fit(self,data,labels):

        N = len(labels)

        for i in range(self.n_estimators):
            if self.bootstrap:
                idxs = np.random.choice(range(N),N)
                data_set = data.iloc[idxs]
                label_set = labels[idxs]
            else:
                data_set = data
                label_set = labels

            root = Node(data_set,label_set)
            root.max_features = self.max_features
            t = Tree(root,self.max_depth,self.min_samples_leaf)

            self.trees.append(t)
```

The parameters for the fit method is a pandas DataFrame and the array of labels. The fit method does not return anything, instead it populates the trees array in the RandomForestObject.

We loop through the n_estimators range.

If bootstrap==True, we make data_set a bootstrap sample from the data, if bootstrap==False we just take data_set to be the data itself

```python
for i in range(self.n_estimators):
    if self.bootstrap:
        idxs = np.random.choice(range(N),N)
        data_set = data.iloc[idxs]
        label_set = labels[idxs]
    else:
        data_set = data
        label_set = labels
```

The predict method takes in an array of data points and return the predictions. It first checks to see if the classifier has been fitted, otherwise it alerts the user.

```python

    def predict(self,data_points):

            if len(self.trees) == 0:
                print('Classifier needs to be fitted')
                return

            preds = []
            for x in data_points:

                x_preds = []
                for t in self.trees:
                    x_preds.append(t.predict(x))

                preds.append(Counter(x_preds).most_common()[0][0])

            return preds


```

For each data point x in the input array, it runs through the trees in the forest and stores the individual tree's prediction in the array x_preds.

We then use Counter to find the majority prediction among all the trees in the forest and appends that to the preds array

```
preds = []
for x in X:
    x_preds = []
    for t in self.trees:
        x_preds.append(t.predict(x))

    preds.append(Counter(x_preds).most_common()[0][0])

return preds
```

Finally we can try out our classifier on a simple data set

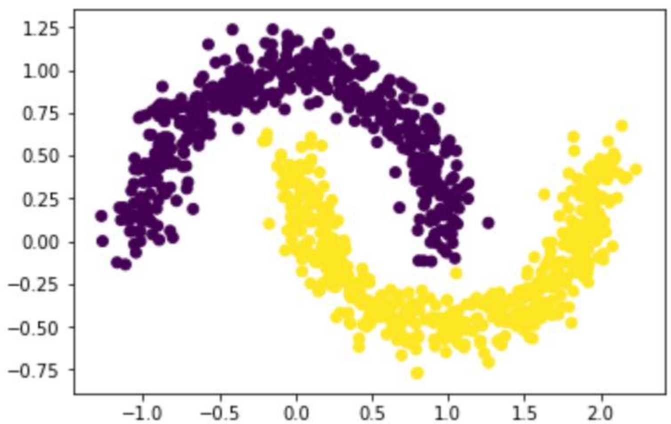
```python
from sklearn.datasets import make_classification
from sklearn.datasets import make_moons
```

```python
X,y = make_moons(1000,noise=1.0)
```

```python
import matplotlib.pyplot as plt
%matplotlib inline
```

```python
plt.scatter(X[:,0],X[:,1],c=y)
```

```
]: <matplotlib.collections.PathCollection at 0x1fdecc50898>
```

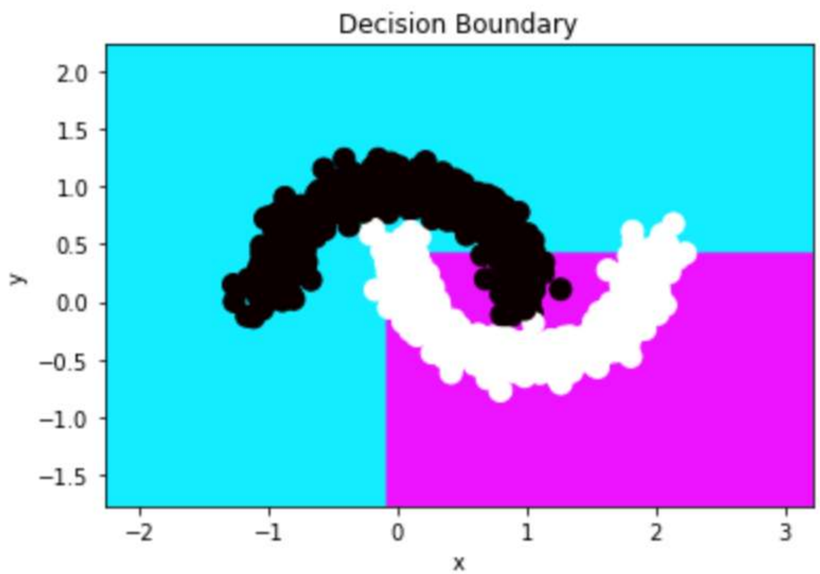



```python
X = pd.DataFrame(data = X, columns = [0,1])
```

```
rf_clf = RandomForestClassifier(n_estimators=20,bootstrap=True,max_features=0.5)
```

```
rf_clf.fit(X,y)
```

```
plot_decision_boundary(X.values,y,rf_clf)
```




```
preds = rf_clf.predict(X.values)
```

```
len(y[y == preds])/1000
```

]: 0.882

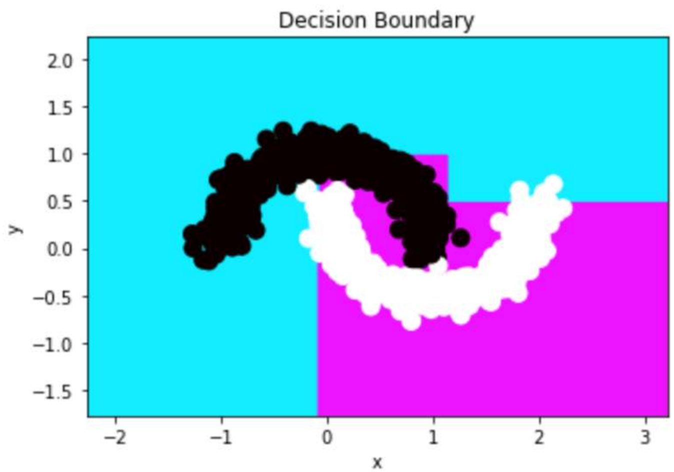
If we change the parameters to limit the depth, we don't get a perfect fit

```
rf_clf = RandomForestClassifier(n_estimators=10,max_features=0.5,bootstrap=True,max_depth=4)
```

```
rf_clf.fit(X,y)
```

```
preds = rf_clf.predict(X.values)
```

```
plot_decision_boundary(X.values,y,rf_clf)
```




```
len(y[y == preds])/1000
```

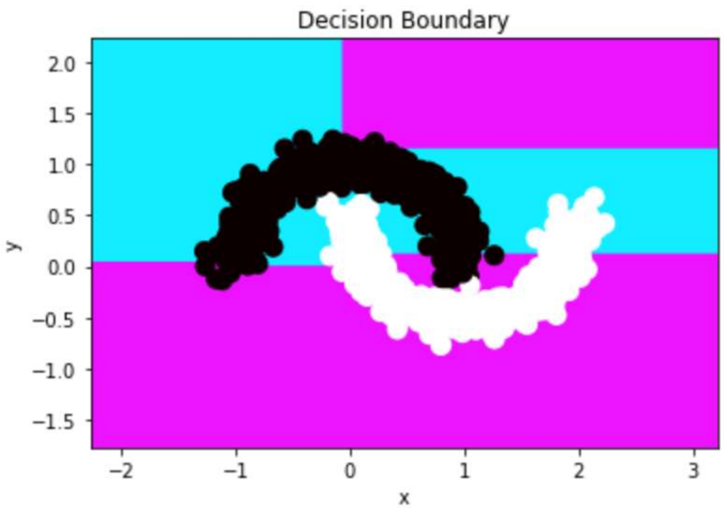
```
0.773
```

```
rf_clf = RandomForestClassifier(n_estimators=10,max_features=1.0,bootstrap=True,min_leaf_samples=15)
```

```
rf_clf.fit(X,y)
```

```
preds = rf_clf.predict(X.values)
```

```
plot_decision_boundary(X.values,y,rf_clf)
```




```
len(y[y == preds])/1000
```

0.85

Remark that setting the min_leaf_samples to a value > 1 limits the depth of the tree.

The depth of an unrestricted tree is at most $\log_2$(size of data) and so the restricted tree has depth at most
$\log_2$(size of data/min_leaf_samples)