Feature importance

What do we have now

- classifier completed, maybe even optimized
- regressor is on the way

Next step : feature importance and print trees

In a decision tree we have, at each node, which feature to choose and the threshold to apply to it in order to follow the left or right branch. One way to interpret a random forest is to measure how important is each of the features. And a simple way to derive importance is to take all the parent nodes from all the trees and count how many times each feature is used.

But feature importance is not the only process that needs to traverse the learned decision trees:

We want also to print all the learned trees of a RF. Again, it is necessary to traverse each tree and

- *if parent, print the feature + threshold and go to children*
- *if leaf, print the prediction and go back to parent*

In the future we forsee other processes will appear that

- *need to traverse the trees in some order and*
- *do something different at a parent and a leaf*

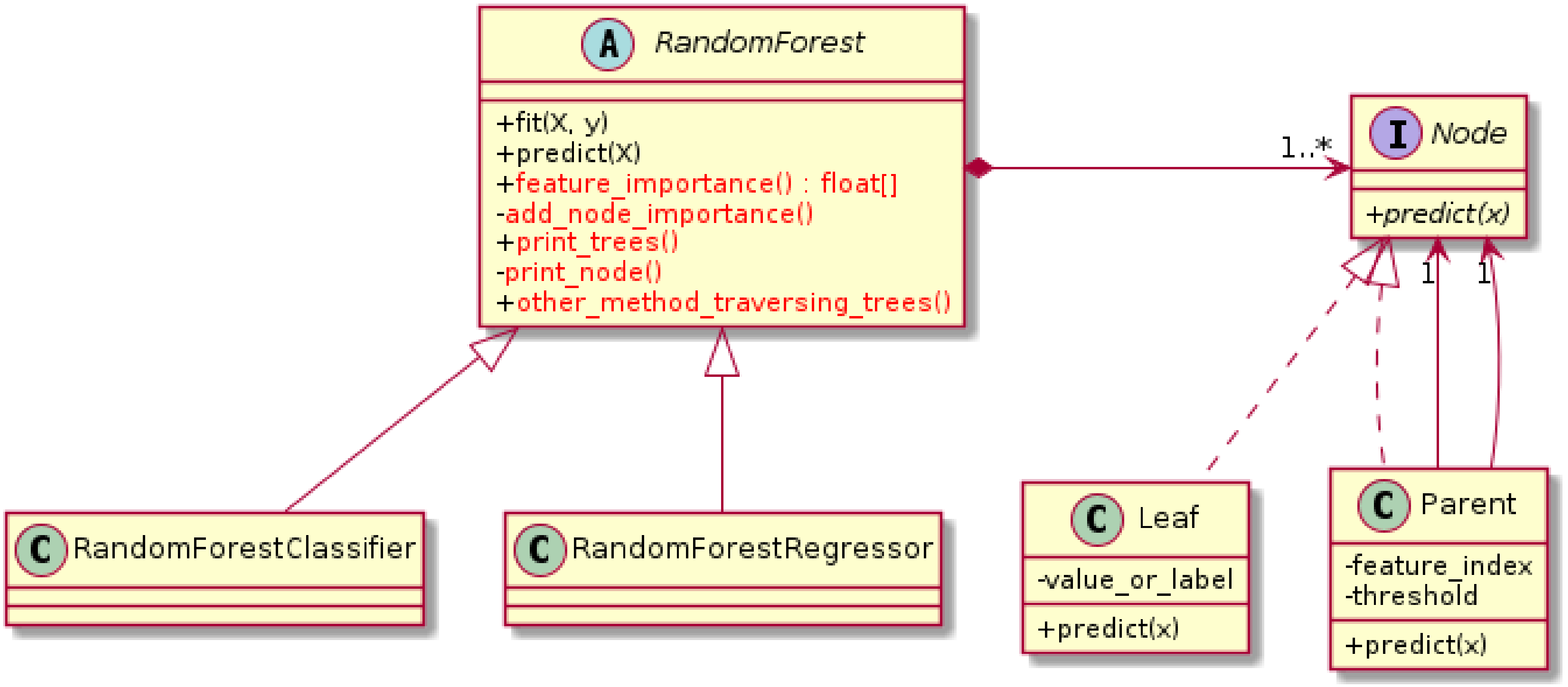
Example of a tree impression

```
decision tree 1
parent, feature index 0, threshold 5.8
        parent, feature index 1, threshold 3.0
                leaf, label or value 1
                parent, feature index 2, threshold 4.5
                        leaf, label or value 0
                        leaf, label or value 1
        parent, feature index 3, threshold 1.8
                parent, feature index 3, threshold 1.0
                        leaf, label or value 0
                        parent, feature index 2, threshold 5.6
                                leaf, label or value 1
                                leaf, label or value 2
                leaf, label or value 2
```

Feature importance the easy way:

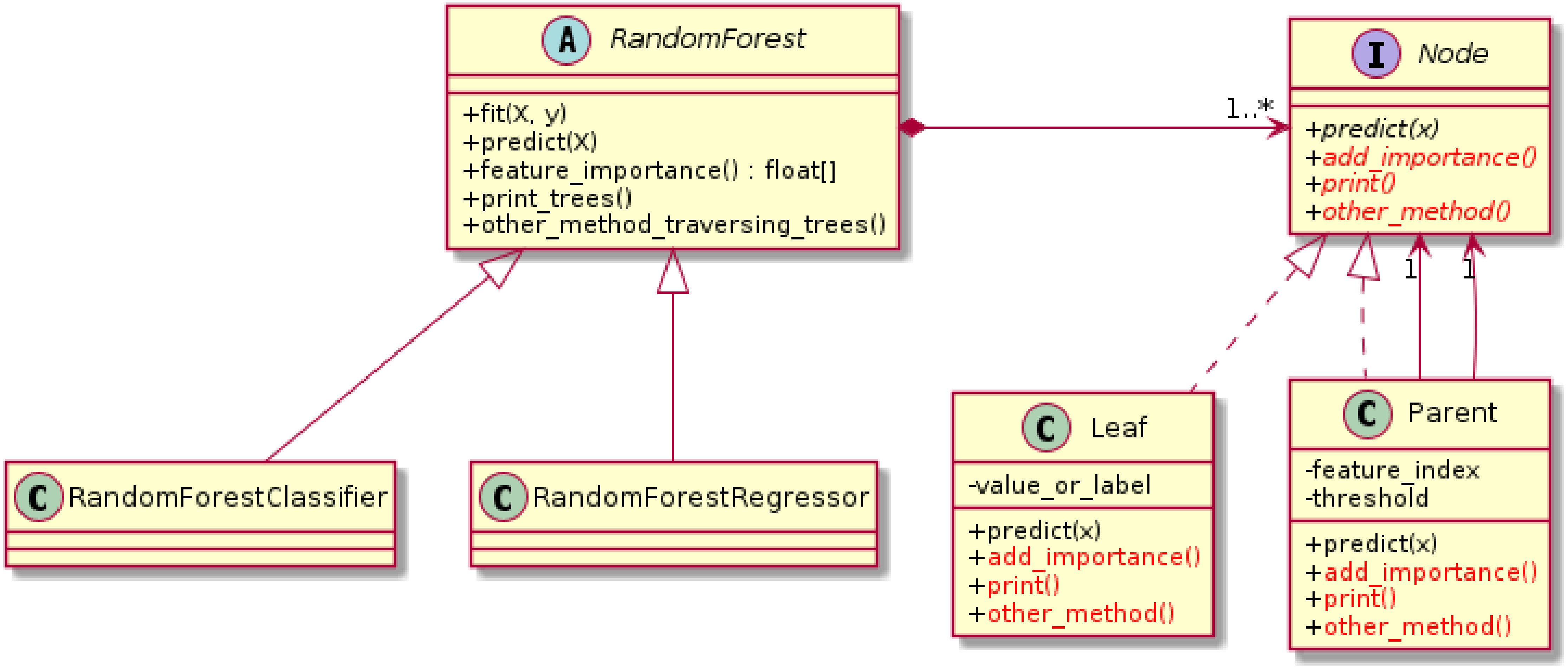
```
def _add_node_importance_(self, node, occurrences):
    if node is a Parent: # doesn't look good, asks for the class of an object
        # preorder recursive
        occurrences[node.idx_feature] += 1
        self._add_node_importance_(node.left_child, occurrences)
        self._add_node_importance_(node.right_child, occurrences)
    else:
        pass # at leaves do nothing

def feature_importance(self, dataset):
    occurrences = {}
    for i in range(dataset.num_features):
        occurences[i] = 0 # dictionary entry
    for tree in self.decision_trees:
        self._add_node_importance_(tree, occurrences) # tree is the root node
```

Solution 1



- new methods needing tree traversal will have to be added to these 2
- `RandomForest` will become more complex (larger) and less cohesive
- take these methods out of `RandomForest`

Solution 2



A RandomForest
+fit(X, y)
+predict(X)
+feature_importance() : float[]
+print_trees()
+other_method_traversing_trees()
1..*
I Node
+predict(x)
+add_importance()
+print()
+other_method()
1
1
C RandomForestClassifier
C RandomForestRegressor
C Leaf
-value_or_label
+predict(x)
+add_importance()
+print()
+other_method()
C Parent
-feature_index
-threshold
+predict(x)
+add_importance()
+print()
+other_method()

- no need to ask for the class of object = good use of inheritance / polymorfism
- each node knows how to add importance, print . . .

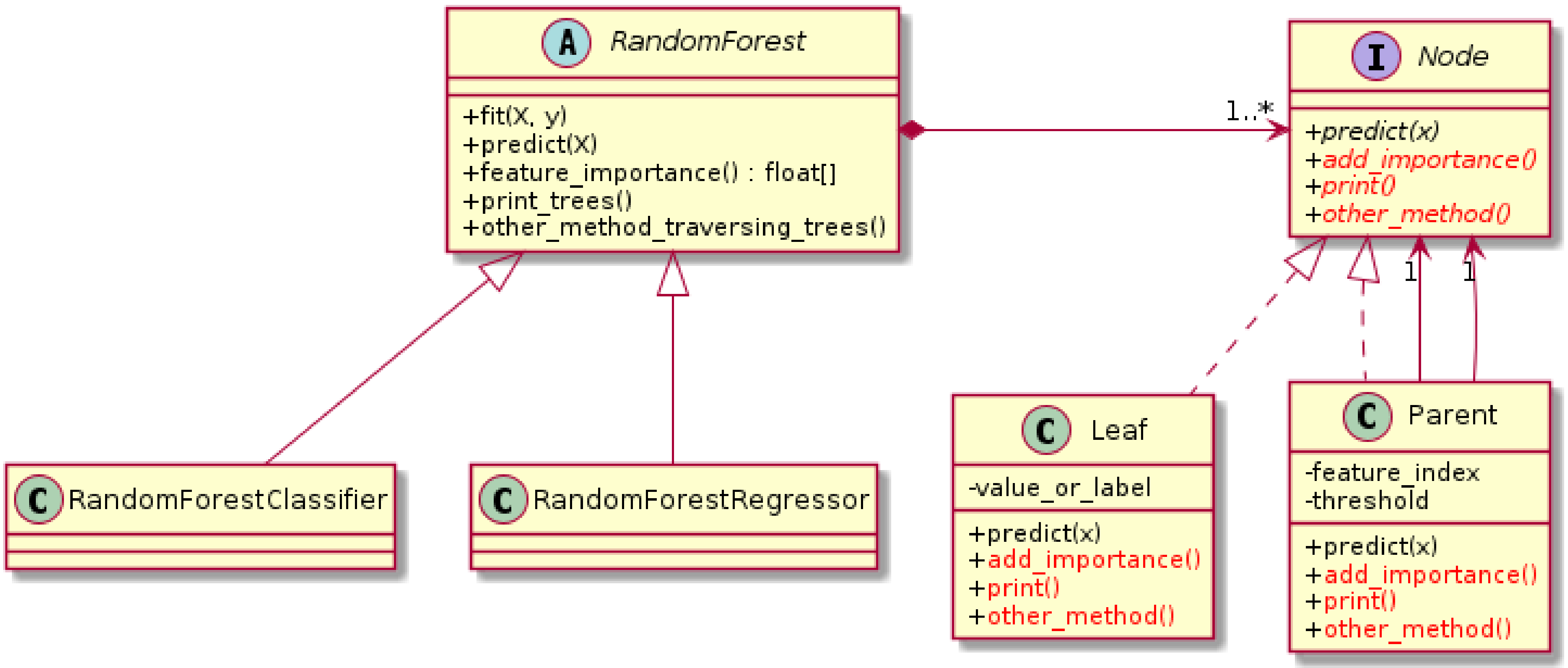
Let's see how to print trees according to this solution

To print all trees we need also to traverse them, in pre-order : first node, then left subtree, then right subtree (recursive)

```python
class Parent(Node):
    def print(self, depth): # preorder, recursive
        print('\t'*depth + 'parent, feature index {}, threshold {}'\
            .format(self.feature_index, self.threshold))
                # '\t'*3 = '\t\t\t', with '\t'=tab
        self.left_child.print(depth+1)
        self.right_child.print(depth+1)

class Leaf(Node):
    def print(self, depth):
        print('\t'*depth + 'Leaf, label or value {}'.format(self.label)

class RandomForest():
    def print_trees(self):
        for tree in self.decision_trees:
            tree.print(depth=0) # tree is the root node
```

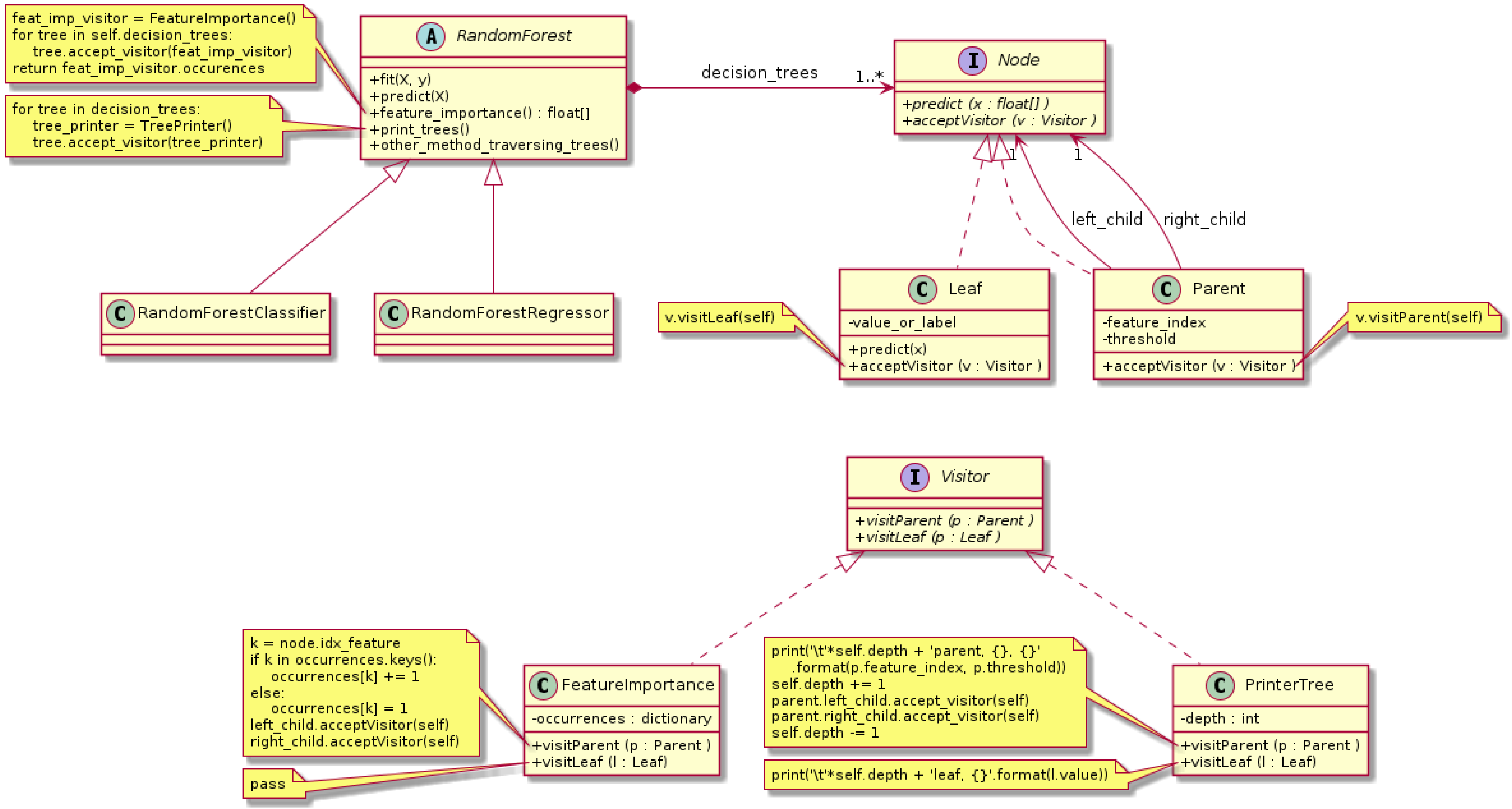
```
decision tree 1
parent, feature index 0, threshold 5.8
        parent, feature index 1, threshold 3.0
                leaf, label or value 1
                parent, feature index 2, threshold 4.5
                        leaf, label or value 0
                        leaf, label or value 1
        parent, feature index 3, threshold 1.8
                parent, feature index 3, threshold 1.0
                        leaf, label or value 0
                        parent, feature index 2, threshold 5.6
                                leaf, label or value 1
                                leaf, label or value 2
                leaf, label or value 2

decision tree 2
parent, feature index 0, threshold 5.5
        parent, feature index 2, threshold 3.0
                leaf, label or value 0
                leaf, label or value 1
        parent, feature index 2, threshold 5.0
                parent, feature index 1, threshold 4.0
                        parent, feature index 2, threshold 4.8
                                leaf, label or value 1
                                parent, feature index 1, threshold 2.8
                                        leaf, label or value 1
                                        leaf, label or value 1
                        leaf, label 0
                parent, feature index 3, threshold 1.8
                        leaf, label or value 2
                        leaf, label or value 2
```

Solution 2 again



- but `Node`, `Leaf`, `Parent` are less cohesive, more complex than before
- we have moved the problem of `RandomForest` to `Node`, `Leaf`, `Parent`

Solution 3



```
feat_imp_visitor = FeatureImportance()
for tree in self.decision_trees:
    tree.accept_visitor(feat_imp_visitor)
return feat_imp_visitor.occurences
```

```
for tree in decision_trees:
    tree_printer = TreePrinter()
    tree.accept_visitor(tree_printer)
```

```
v.visitLeaf(self)
```

```
v.visitParent(self)
```

```
k = node.idx_feature
if k in occurrences.keys():
    occurrences[k] += 1
else:
    occurrences[k] = 1
left_child.acceptVisitor(self)
right_child.acceptVisitor(self)
```

```
pass
```

```
print('\t'*self.depth + 'parent, {}, {}'
    .format(p.feature_index, p.threshold))
self.depth += 1
parent.left_child.accept_visitor(self)
parent.right_child.accept_visitor(self)
self.depth -= 1
```

```
print('\t'*self.depth + 'leaf, {}'.format(l.value))
```

- now all classes are cohesive
- we can add new functionalities that need to traverse trees just by creating new visitors
- note that each visitor defines its own order, here both preorder
- but design is more complex, need to know visitor pattern

Expected results

Iris

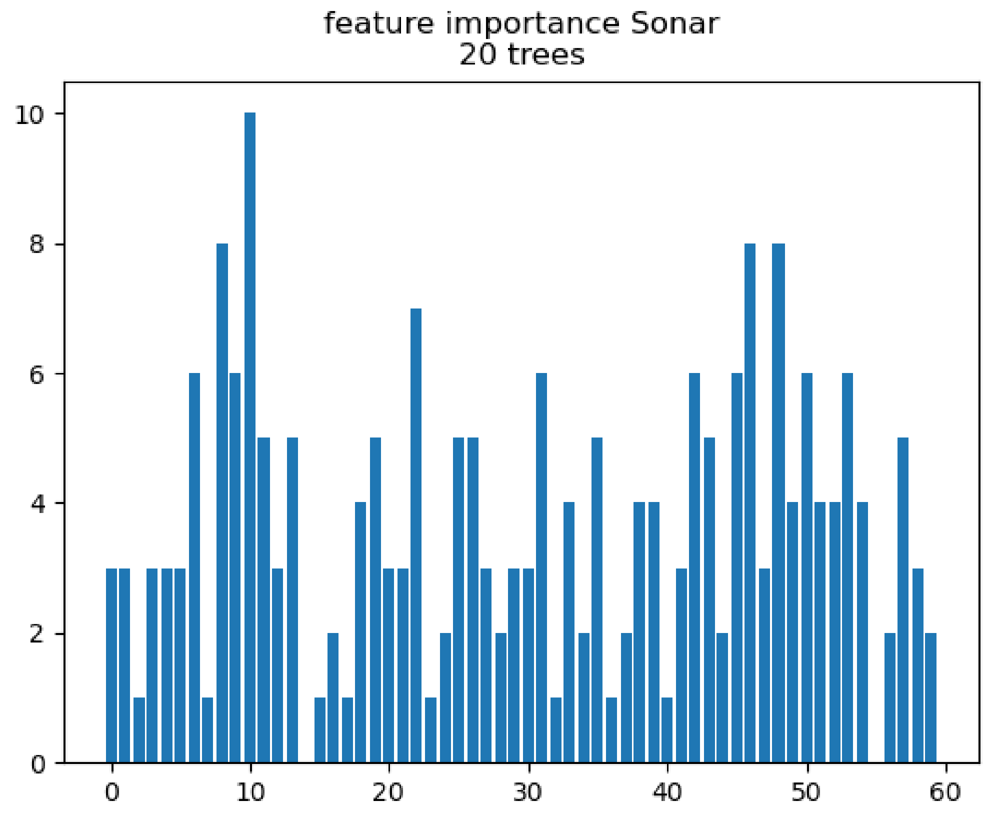
```
# fit a RF classifier with Iris train dataset
occurrences = rf.feature_importance()
print('Iris occurrences for {} trees'.format(rf.num_trees))
print(occurrences)
```

Iris : `occurrences = {3: 33, 2: 30, 0: 12, 1: 8}`

3,2 = sepal length, width

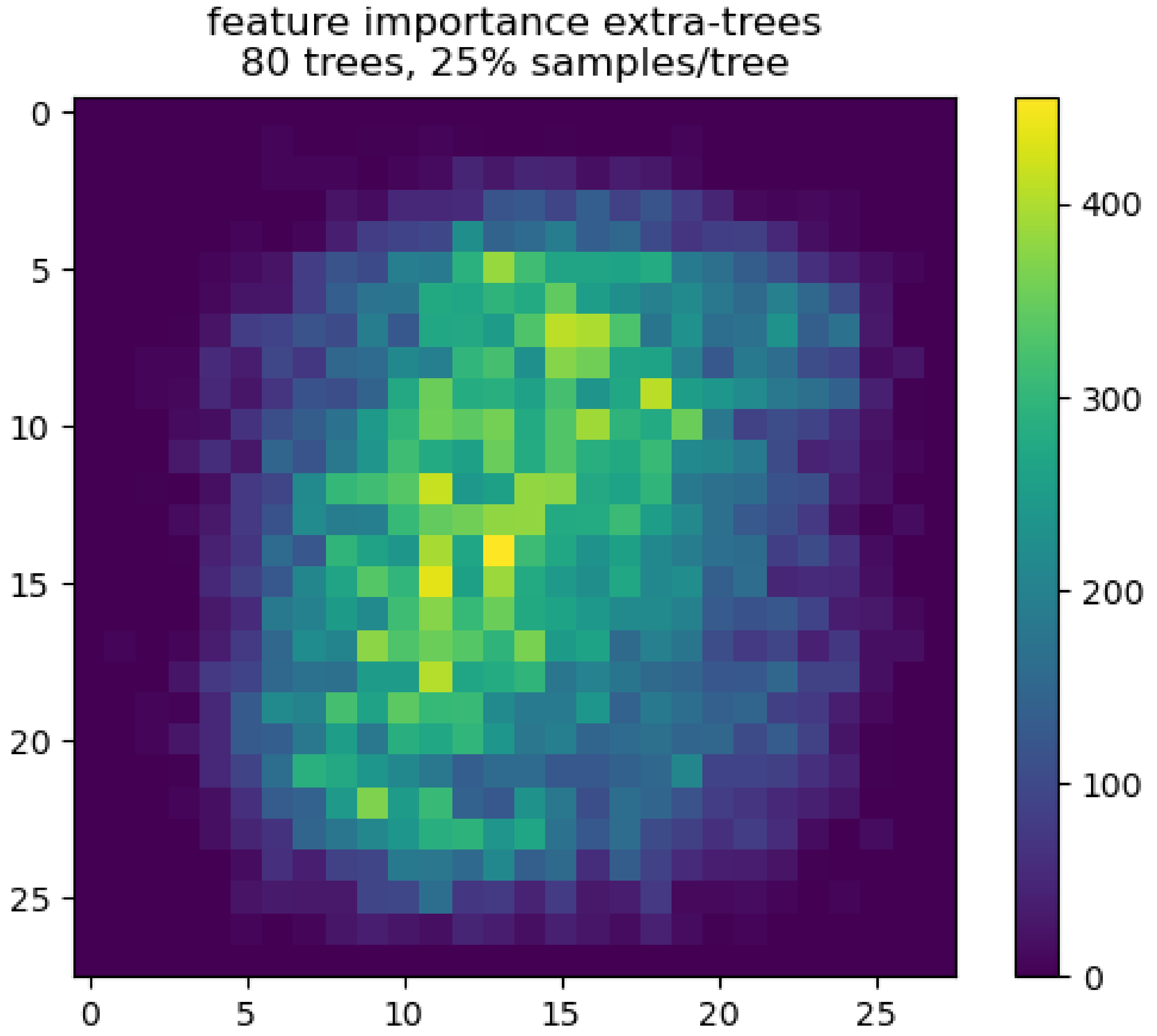
Sonar

```
# fit a RF classifier with sonar train dataset
occurrences = rf.feature_importance() # a dictionary
counts = np.array(list(occurrences.items()))
plt.figure(), plt.bar(counts[:, 0], counts[:, 1])
plt.xlabel('feature')
plt.ylabel('occurrences')
plt.title('Sonar feature importance\n{} trees'.format(rf.num_trees))
```



MNIST

```
if not os.path.exists('rf_mnist.pkl'):
    rf = # train an optimized RF classifier with MNIST train dataset
    # max_depth=20, min_size=20, ratio_samples=0.25, num_trees=80,
    # num_random_features=28, impurity='gini',
    # do_multiprocessing=True, optimization='extra-trees'
    with open('rf_mnist.pkl', 'wb') as f:
        pickle.dump(rf, f)
        # save the rf object so we won't need to train again
else:
    with open('rf_mnist.pkl','rb') as f:
        rf = pickle.load(f)
occurrences = rf.feature_importance()
ima = np.zeros(28*28)
for k in occurrences.keys():
        ima[k] = occurrences[k]
plt.figure()
plt.imshow(np.reshape(ima,(28,28)))
plt.colorbar()
plt.title('Feature importance MNIST')
```


feature importance extra-trees
80 trees, 25% samples/tree
0
5
10
15
20
25
0
5
10
15
20
25
400
300
200
100
0