Random Forest algorithm

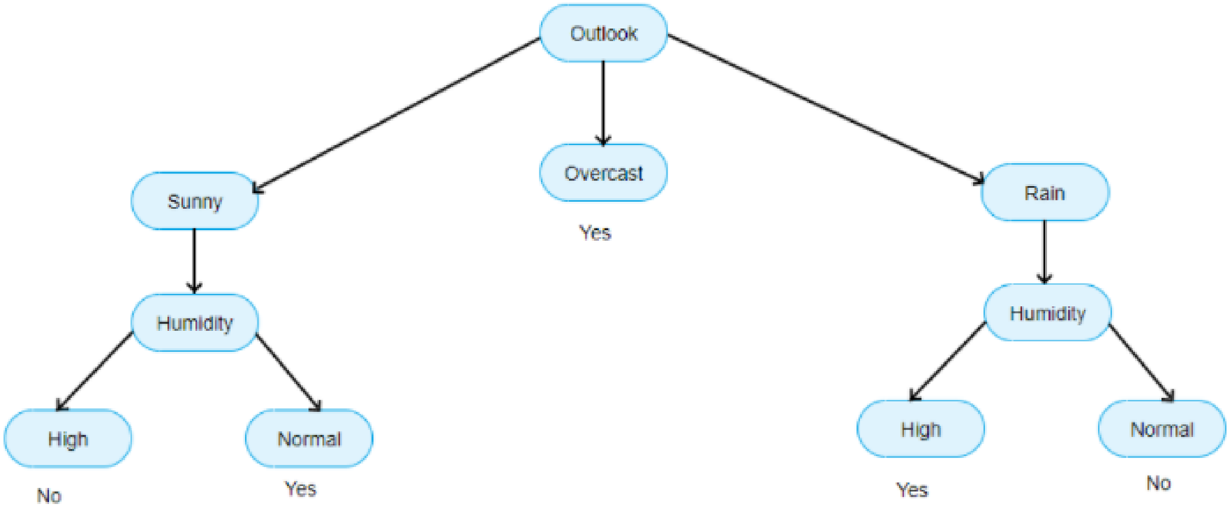
Introduction

Implementation of Random Forest in Python and Go. Both sequential and parallel versions are developed. Visualization is done using Pharo and Roassal library. We'll compare the results based on increased datasets and execution time as well as increased number of Decision Trees and execution time.

Definition

This project is focused on Random Forests for classification purposes. Random Forest is an ensemble method. It is a form of bagging (bootstrap aggregating) algorithm which means that it consists of **multiple weak classificators** and makes a final prediction based on the majority of votes. It's main advantage is that it reduces variance - decreases overfitting. The base estimator for Random Forest is the Decision Tree.

Decision Tree



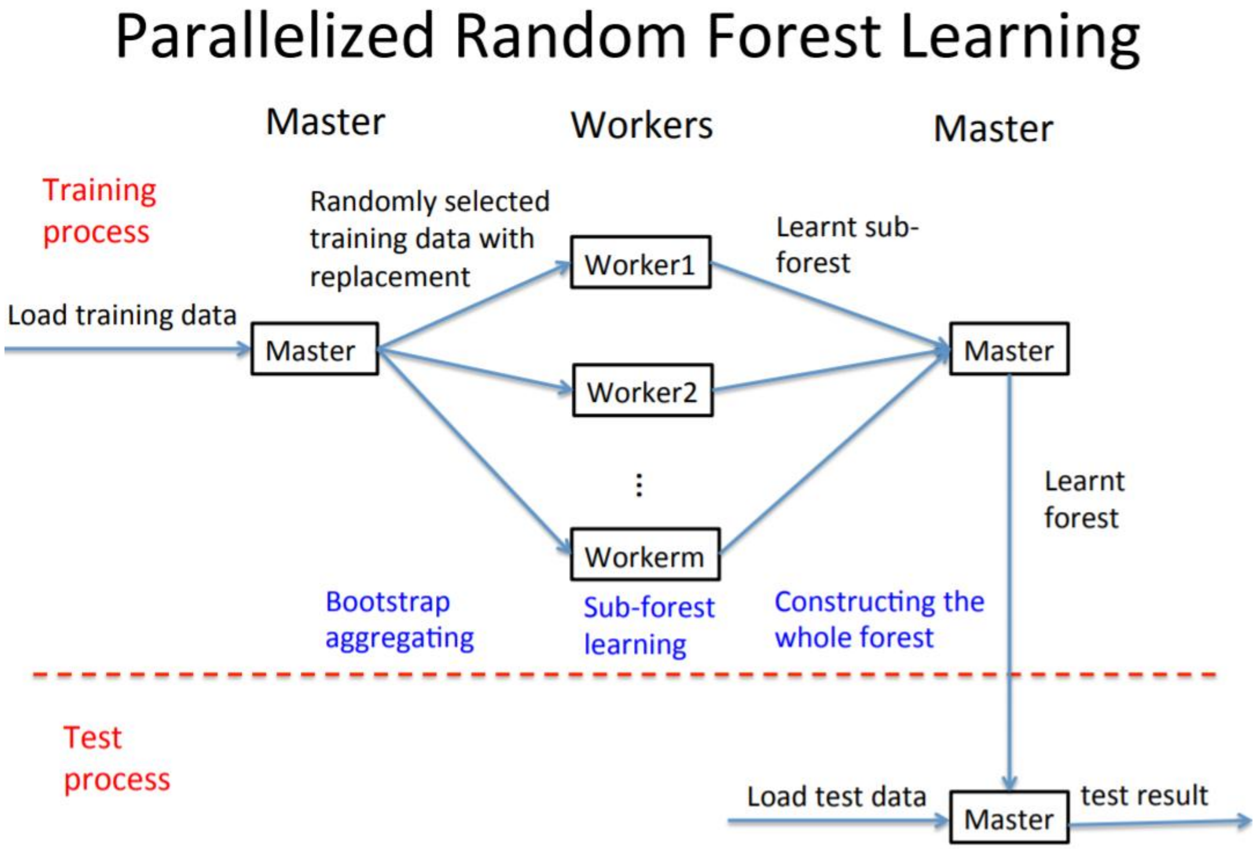
source: *https://www.aitimejournal.com/@akshay.chavan/a-comprehensive-guide-to-decision-tree-learning*

The tree is built with the root note, decision nodes and leaf nodes. Root node and decision nodes consist of conditions. We start with each example and run it through the tree, checking the conditions and navigating the tree accordingly. The final prediction is stored in the leaf node.

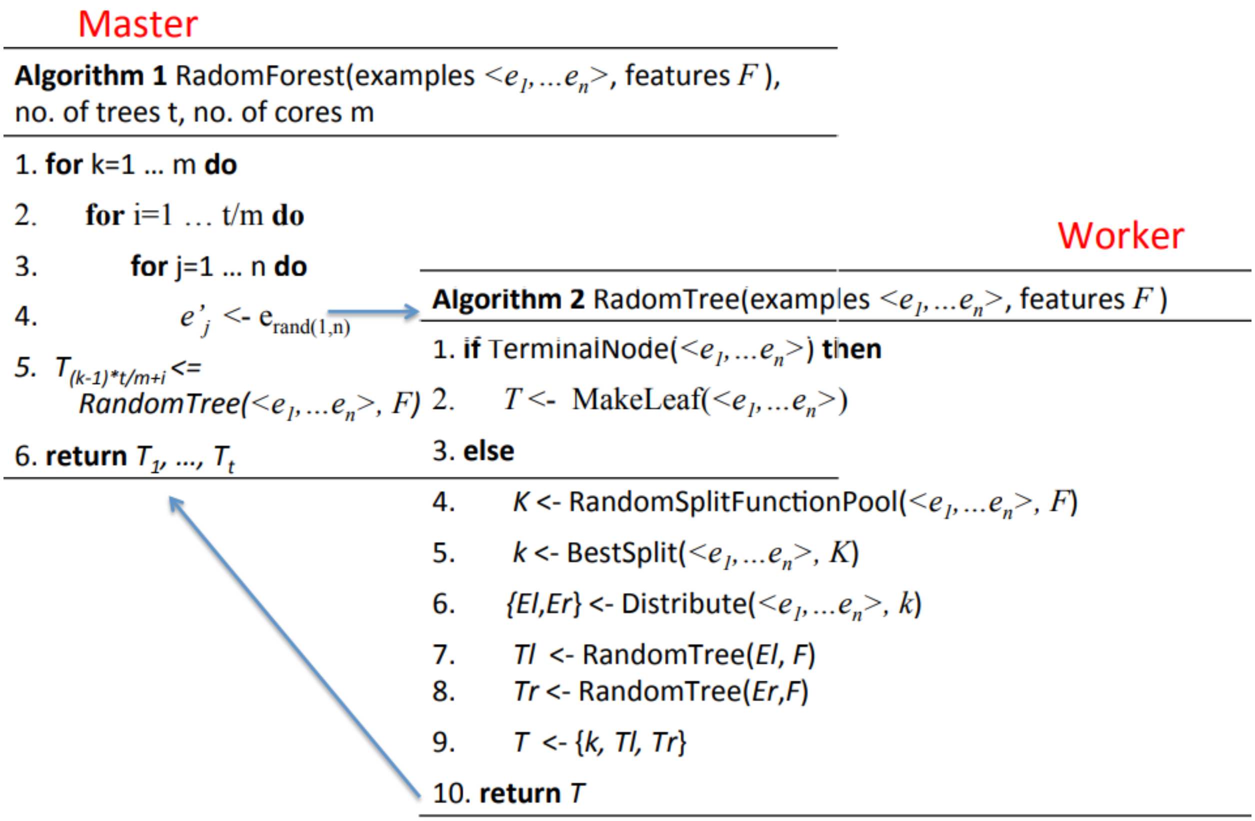
Parallelization

As Random Forest consists of multiple weak classificators it is a great candidate for paralellization. (**bagging** generally is great for this purpose) Every classificator is run on the subset of data on different threads. As we have larger and larger datasets, the advantage of parallel approach is obvious.

- The complexity of sequential version is O(t * v * n * logn)
    - t is number of trees
    - v is number of features we selected
    - n is number of samples we take into consideration



Pseudo code for the parallel version is listed below. We're making bagged subsets of our dataset and group trees in batches that we train on multiple threads.



Master

```
Algorithm 1 RadomForest(examples $<e_1, \dots e_n>$, features $F$ ), no. of trees t, no. of cores m
for k=1 ... m do
    for i=1 … t/m do
        for j=1 ... n do
            $e'_j$ <- $e_{rand(1,n)}$
T_(k-1)*t/m+i <= RandomTree($<e_1, \dots e_n>$, $F$)
return $T_1$, ..., $T_t$
```

Worker

```
Algorithm 2 RadomTree(examples $<e_1, \dots e_n>$, features $F$ )
if TerminalNode($<e_1, \dots e_n>$) then
    $T$ <- MakeLeaf($<e_1, \dots e_n>$)
else
    $K$ <- RandomSplitFunctionPool($<e_1, \dots e_n>$, $F$)
    $k$ <- BestSplit($<e_1, \dots e_n>$, $K$)
    {$El$,$Er$} <- Distribute($<e_1, \dots e_n>$, $k$)
    $Tl$ <- RandomTree($El$, $F$)
    $Tr$ <- RandomTree($Er$,$F$)
    $T$ <- {$k$, $Tl$, $Tr$}
return $T$
```

Hardware Specification

Processor: Intel Core i5 4300 @ 1.90 GHz 2.50 Ghz 2 cores

RAM: 8 GB DD3

MEMORY: 256GB SSD

OS: Windows 10 Professional 64bit

Libraries used:

- Python
  - Numpy 1.16.2
  - Pandas 0.24.2
- Go
  - Dataframe
  - gonum/mat v1

Strong Scaling & Amdahl's law

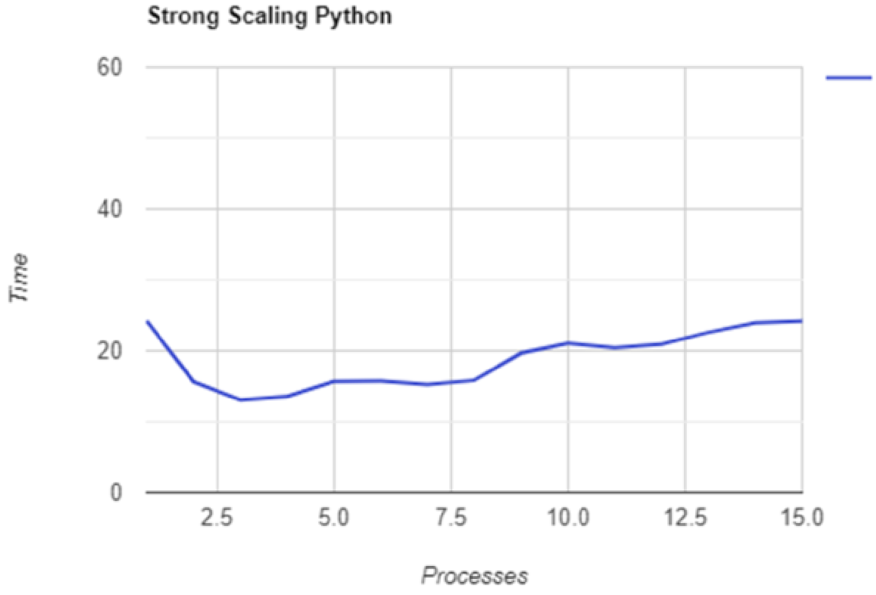
Law of scalability by Amdahl is:

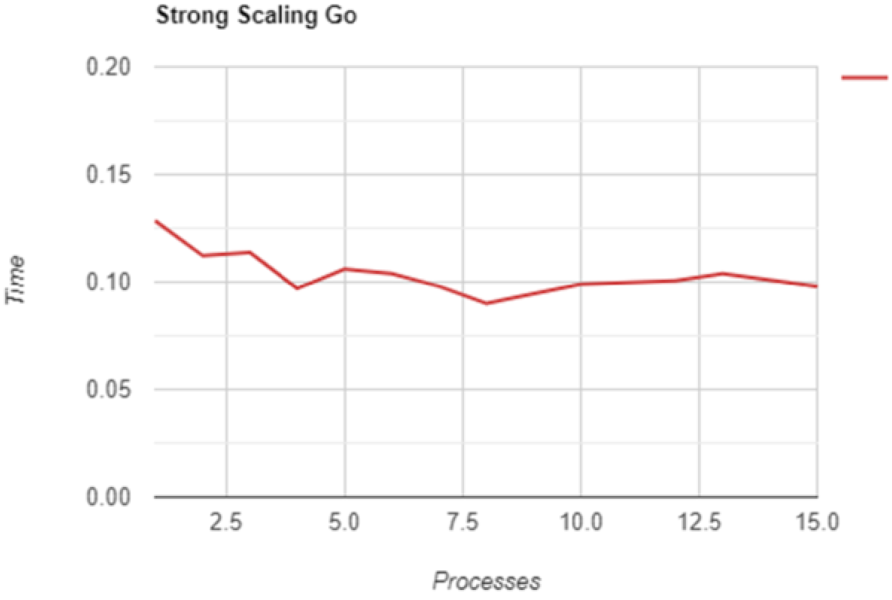
Speedup = 1 / (**s** + **p**/N)

Where ***s*** is the proportion of execution time spent on sequential part and ***p*** is the proportion of execution spent on the parallel part. On the other hand the ***N*** is the number of processors we're running the program on. In Random Forest we are not parallelizing a huge chunk of code. We're only delegating bunch of trees training. Estimated S in this project = 70% and P the rest 30%.

The results achieved are in the table below.

Processes	**Time**		
	PYTHON	GO	Speedup
1	24.2384	0.1284	1.14
2	15.6644	0.11228	1.14
3	13.0808	0.1137	1.14
4	13.5752	0.097	1.31
5	15.6794	0.106	1.19
6	15.7383	0.104	1.19
7	15.2666	0.098	1.30
8	184896	0.09	1.42
9	19.8672	0.099	1.29
10	21.0597	0.1006	1.27
11	20.4560	0.104	1.23
12	20.9255	0.098	1.30
13	22.5914	0.096	1.33
14	23.9325	0.0960	1.33
15	24.2067	0.0909	1.42





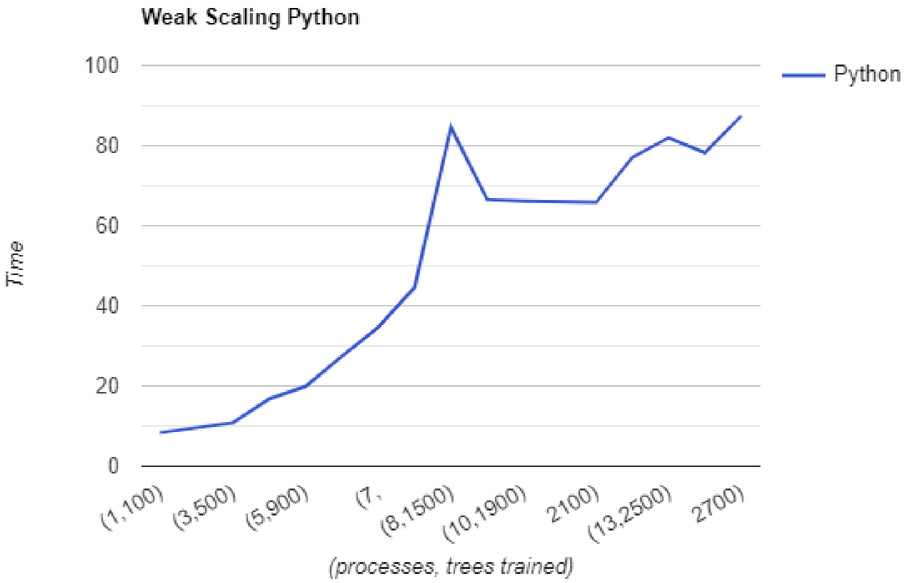
In both examples we've used 700 trees for the training phase. We conclude that Go is much faster- it is a compiled language as opposed to Python. Interestingly we see the increase in Python's execution time as the number of processes increase. Bear in mind the processor on the testing machine is very weak. The cost of creating processes becomes larger.

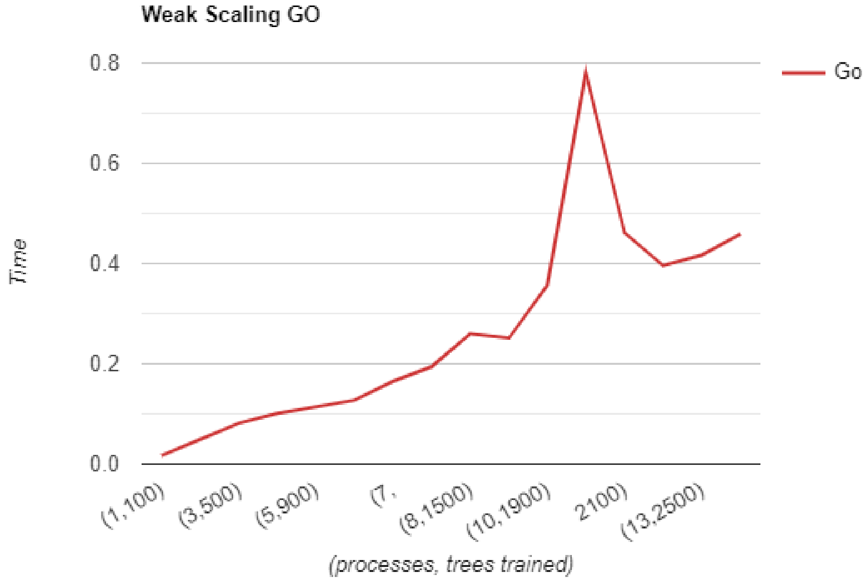
Weak Scaling & Gustafson's law

On the other hand we have the Gustafson's law that states

Speedup = s + p * N

As we've estimated S ~ 70% and P ~ 30% we can see that for our 2 core processor we can get x1.3 speedup theoretically. The experiments of weak scaling are listed below.


Weak Scaling Python
Python
Time
(1,100)
(3,500)
(5,900)
(7,
(8,1500)
(10,1900)
2100)
(13,2500)
2700)
(processes, trees trained)


Weak Scaling GO
Go
Time
(1,100)
(3,500)
(5,900)
(7,
(8,1500)
(10,1900)
2100)
(13,2500)
(processes, trees trained)